**- Hour 1 -**

**Introducing Visual C++ 5**

Welcome to Hour 1 of *Teach Yourself Visual C++ 5 in 24 Hours*! Visual C++ is an exciting subject, and this first hour gets you right into the basic features of the new Visual C++ 5 compiler and starts you off building some basic programs.

These are the highlights of this hour:

• A short overview of the Visual C++ environment and how to work in it

• How to compile a simple console-mode program

• How to use AppWizard to create a Windows application

**Exploring Visual C++ 5**

Visual C++ 5 is the latest C++ compiler from Microsoft, continuing a long line of Microsoft tools for Windows development. The Visual C++ package contains more than a compiler; it also contains all the libraries, examples, and documentation needed to create applications for Windows 95 and Windows NT.

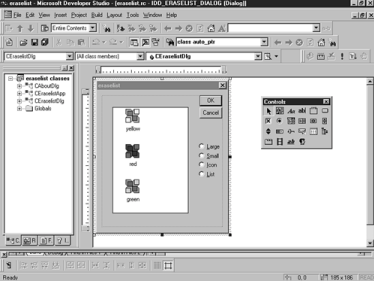
Windows development tools have certainly come a long way since the earliest C and C++ compilers for Windows. By combining into a single tool all the resources required to build Windows applications, Microsoft has made it much easier for you to learn to build applications.

**The Visual C++ Environment**

**New Term:** An *IDE*, or *Integrated Development Environment*, is a program that hosts the compiler, debugger, and application-building tools.

The central part of the Visual C++ package is *Developer Studio*, the Integrated Development Environment (IDE), shown in Figure 1.1. Developer Studio is used to integrate the development tools and the Visual C++ compiler. You can create a Windows program, scan through an impressive amount of online help, and debug a program without leaving Developer Studio.

1

**Figure 1.1.** Using Developer Studio to create a Windows program.

Visual C++ and Developer Studio make up a fully integrated environment that makes it very easy to create Windows programs. By using the tools and wizards provided as part of Developer Studio, along with the MFC class library, you can create a program in just a few minutes.

Many of the programs used as examples in this book require less than a page of additional source code. However, these programs use the thousands of lines of source code that are part of the MFC class library. They also take advantage of AppWizard and ClassWizard, two of the Developer Studio tools that manage your project for you.

**Developer Studio Tools**

Once upon a time, Windows programmers used simple text editors and tools that were hosted on MS-DOS to create their Windows programs. Developing a program under those conditions was tedious and error-prone. Times have definitely changed; Developer Studio includes a number of tools that you might once have paid extra to purchase.

• An integrated editor offers drag-and-drop and syntax highlighting as two of its major features. You can configure the Developer Studio editor to emulate the keystroke commands used by two popular programmer's editors, Brief and Epsilon.

2

• A resource editor is used to create Windows resources, such as bitmaps, icons, dialog boxes, and menus.

• An integrated debugger enables you to run programs and check for errors. Because the debugger is part of Developer Studio, it's easy to find and correct bugs. If you find a programming error while debugging, you can correct the source code, recompile, and restart the debugger.

Developer Studio also features an online help system, which can be used to get context-sensitive help for all of the tools included in Developer Studio, as well as detailed help on the C++ language, the Windows programming interface, and the MFC class library.

**Developer Studio Wizards**

**New Term:** A *Wizard* is a tool that helps guide you through a series of steps.

In addition to tools that are used for debugging, editing, and creating resources, Developer Studio includes several wizards that are used to simplify developing your Windows programs. The most commonly used ones are

• *AppWizard* (also referred to in some screens as MFC AppWizard) is used to create the basic outline of a Windows program. Three types of programs are supported by AppWizard: single document and multiple document applications based on the Document/View architecture and dialog box-based programs, in which a dialog box serves as the application's main window. Later in this hour, you will use AppWizard to create a simple program.

• *ClassWizard* is used to define the classes in a program created with AppWizard. Using ClassWizard, you can add classes to your project. You can also add functions that control how messages received by each class are handled. ClassWizard also helps manage controls that are contained in dialog boxes by enabling you to associate an MFC object or class member variable with each control. You will learn more about ClassWizard in Hour 4, "Dialog Boxes and C++ Classes."

• *ActiveX ControlWizard* is used to create the basic framework of an ActiveX control. An ActiveX control is a customized control that supports a defined set of interfaces and is used as a reusable component. ActiveX controls replace Visual Basic controls, or VBXs, which were used in 16-bit versions of Windows. ActiveX controls are used in Hour 20, "Using ActiveX Controls," and you will build an ActiveX control in Hour 24, "Creating ActiveX Controls."

**MFC Libraries**

**New Term:** A *library* is a collection of source code or compiled code that you can reuse in your programs. Libraries are available from compiler vendors such as Microsoft, as well as from third parties.

**New Term:** Visual C++ 5 includes Version 5.0 of *MFC*, the Microsoft Foundation Classes, a class library that makes programming for Windows much easier.

3

By using the MFC classes when writing your programs for Windows, you can take advantage of a large amount of source code that has been written for you. This enables you to concentrate on the important parts of your code rather than worry about the details of Windows programming.

**New Term:** A recent addition to the C++ standard is the *Standard C++ Library*. This library includes a set of classes that were known as the Standard Template Library, or STL, during the standardization process. Unlike the MFC class library, which is used primarily for Windows programming, the standard C++ library is used for general-purpose programming.

**Starting Developer Studio**

To start Developer Studio, click the Developer Studio icon located in the Visual C++ folder. To get to the Visual C++ folder, click the Start button on the taskbar and then select Programs. One of the items in the Programs folder is Microsoft Visual C++ 5.0. Figure 1.2 shows a start menu tree opened to the Microsoft Developer Studio icon.

**Figure 1.2.** Starting Developer Studio from the Start button.

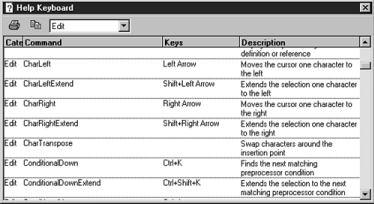
Developer Studio initially displays two windows:

• A Project Workspace window located on the left side; this window contains a table of online help contents

4

• A Document window on the right side; this window contains the documentation home page

Developer Studio also includes a rich set of menus, toolbars, and other user interface features, as shown in Figure 1.3.

**Figure 1.3.** *Developer Studio when first started.*

**Exploring InfoViewer**

InfoViewer is the online help system integrated into Developer Studio. InfoViewer is also compatible with the Microsoft Developer Network CD-ROM, enabling you to search that database for information.

**Time Saver:** Usually, the indexes used by the InfoViewer are copied to your hard disk and the actual database remains on the CD-ROM. If you would like to speed up InfoViewer, run Visual C++ setup again and install InfoViewer to the hard disk.

**Using Dockable Windows in Developer Studio**

**New Term:** Many of the views displayed by Developer Studio are *dockable*, which means they can be attached to the edge of the Developer Studio workspace, where they remain until undocked.

The Project Workspace window shown in Figure 1.3 is an example of a dockable view. To "undock" a dockable window, double-click the window's edge. To dock a floating window, move it to the edge of the workspace. If it is a dockable window, it docks itself. If you want to move a dockable window close to the edge of a workspace without docking, press the Ctrl key on the keyboard when moving the window.

5

**Getting Context-Sensitive Help**

To get context-sensitive help from InfoViewer, press F1. You select a topic based on the current window and cursor position, and you see the InfoViewer window, containing context-sensitive help. If you press F1 while editing a source file, help is provided for the word under the cursor. If there is more than one possible help topic, you see a list of choices.

**The Visual C++ Editor**

Developer Studio includes a sophisticated editor as one of its tools. The editor is integrated with the other parts of Developer Studio; files are edited in a Developer Studio child window.

You use the Developer Studio editor to edit C++ source files that will be compiled into Windows programs. The editor supplied with Developer Studio is similar to a word processor, but instead of fancy text-formatting features, it has features that make it easy to write source code.

You can use almost any editor to write C++ source code, but there are several reasons to consider using the editor integrated with Developer Studio. The editor includes many features that are found in specialized programming editors.

• Automatic syntax highlighting colors keywords, comments, and other source code in different colors. • Automatic "smart" indenting helps line up your code into easy-to-read columns. • Emulation for keystrokes used by other editors helps if you are familiar with editors such as Brief and Epsilon.

• Integrated keyword help enables you to get help on any keyword, MFC class, or Windows function just by pressing F1.

• Drag-and-drop editing enables you to move text easily by dragging it with the mouse. • Integration with the compiler's error output helps you step through the list of errors reported by the compiler and positions the cursor at every error. This enables you to make corrections easily without leaving Developer Studio.

**Just a Minute:** If you do choose to use another editor to create your source files, make sure the files are stored as ASCII, also known as "plain text" files. The Visual C++ compiler cannot process files that have special formatting characters embedded in them, such as the files created by word- processing programs.

**Using Editor Commands**

A large set of editing commands are available from the keyboard. Although most editor commands are also available from the menu or toolbar, the following commands are frequently used from the keyboard:

6

• *Undo*, which reverses the previous editor action, is performed by pressing Ctrl+Z on the keyboard. The number of undo steps that can be performed is configurable in the Options dialog box. • *Redo*, which is used to reverse an undo, is performed by pressing Ctrl+Y.

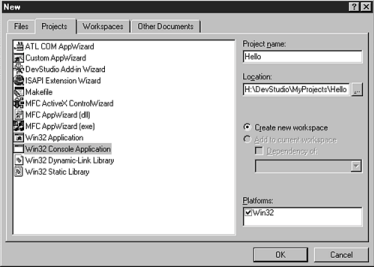
• *LineCut*, which removes or "cuts" the current line and places it on the Clipboard, is performed by pressing Ctrl+L.

• *Cut* removes any marked text from the editor and places it on the Clipboard. This command is performed by pressing Ctrl+X.

• *Copy* copies any marked text to the Clipboard but, unlike the Cut command, doesn't remove the text from the editor. If no text is marked, the current line is copied. This command is performed by pressing Ctrl+C.

• *Paste* copies the Clipboard contents into the editor at the insertion point. This command is performed by pressing Ctrl+V.

This is only a small list of the available keyboard commands. To see a complete list, select Keyboard Map... from the Help menu. A list of the current keyboard command bindings is displayed, as shown in Figure 1.4.

**Figure 1.4.** An example of keyboard command bindings in Developer Studio.

7

**Creating Your First C++ Program**

**New Term:** A *console-mode* application is a character-based program that runs in a DOS window.

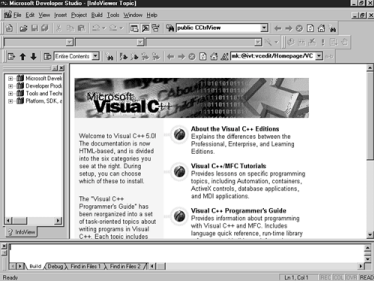
For your first Visual C++ program, you will build a console-mode program that displays a Hello World greeting. Console-mode programs are often simpler to build than Windows applications, and this example will take you through the steps of building and running a program built with Visual C++.

**Starting Your First Program**

The first stage in writing your first Visual C++ program is to create a project. Follow these steps:

1. Choose File|New from the main menu. The New dialog box will be displayed. 2. Select the Projects tab, and then click the Win32 Console Application icon from the list box.

3. Specify Hello as the project name in the Project name box; a default location for your project will automatically be entered in the Location box (see Figure 1.5).

**Figure 1.5.** The New Projects dialog box for the Hello project.

4. Click OK to create the project.

8

**Editing Your First C++ Source File**

The most important parts of any C++ program are the source files. Although the sample program provided in Listing 1.1 is very short, it contains many of the elements present in all C++ programs.

**TYPE: Listing 1.1. A simple C++ console-mode program.**

// Hello world example

#include <iostream>

using namespace std;

int main()

{

cout << "Hello World!" << endl;

return 0;

}

Open a new source file document and type the program exactly as shown in Listing 1.1. As discussed earlier, there are two ways to open a new source file for editing:

• Click the New Text File icon on the toolbar.

• Select File|New from the main menu, and select C++ Source File from the New dialog box under the Files tab.

If you open a new file for editing while a project is open, you have the option of automatically adding the file to the currently open project. To take advantage of this option, make sure the Add to Project: check box is checked, and provide a name for the file in the dialog box (in this case use Hello.cpp).

**CAUTION:** When using C++, remember that capitalization is important. For example, MAIN and main are two different names to the compiler. White space, such as the number of spaces before a word such as cout, is not significant to the compiler. White space is often used to help make programs more readable.

**Just a Minute:** If you used the toolbar's New Source File icon to create your new source file, syntax highlighting will not be provided until the file is saved and the file is given a name. This is because the Developer Studio editor uses the file extension to determine the file type, and it does not know what type of file is being edited.

9

**Saving a Source File**

After you have entered the program in Listing 1.1, save the source file in your project's directory as Hello.cpp. To save the contents of the editor, click the Save icon on the toolbar. The Save icon looks like a small floppy disk. You can also press Ctrl+S or select Save from the File menu.

When updating a previously saved source file, you don't see a dialog box, and no further action is needed on your part. The existing file is updated using the current contents of the editor. If you save a new file, you see the Save As dialog box, and you must choose a location and filename for the new source file. Save the contents of Listing 1.1 in the C:\ directory using the name CFoo.cpp. After saving the file, close CFoo.cpp by selecting Close from the File menu.

To save a file under a new name, select Save As from the File menu or press F12. Enter the new path and filename using the Save As dialog box as described previously.

If you have not yet added the source file to the project, follow these steps:

1. Select Project|Add To Project|Files... from the main menu. This will display the Insert Files into Project dialog box.

2. Select the Hello.cpp source file and then click OK.

**Just a Minute**: Visual C++ requires that your C++ source files have a .CPP file extension. This helps Developer Studio properly compile your source code, as well as provide the proper syntax highlighting.

Other types of files also have standard extensions. For example, C source files must use the .C extension. Other file extensions will be discussed as they are introduced.

**Building the Hello Project**

Compile the Hello project by selecting Build|Build Hello.exe from the main menu (or press F7). If you entered Listing 1.1 correctly, the project is built with no errors, and the last line in the status window reads as follows:

HELLO.exe - 0 error(s), 0 warning(s)

**Time Saver:** You can also build the Hello project by clicking the Build button on the toolbar. The toolbar was shown in Figure 1.3.

If errors or warnings are displayed in the Build status window, there is probably an error in the source file. Check your source file again for missing semicolons, quotes, or braces.

10

**Running Your First C++ Program**

To run the Hello program, open a DOS window and change the working directory to the project's directory. By default, this directory is

C:\Program File\DevStudio\MyProjects\Hello

On some machines, filenames may be truncated, so the path on your machine might be something like C:\progra~1\devstudio\myprojects\hello

You'll see a subdirectory named DEBUG. The Visual C++ IDE puts all the executable and intermediate files into this directory by default. Change to the DEBUG directory and execute the Hello.exe program by typing the following at the DOS prompt:

HELLO

The program loads and then displays Hello World!. That's all there is to it.

All of the console mode or DOS programs used as examples in this book should be compiled and executed just like Hello.exe. You'll always create a project, add files to it, and then build the project. After the application is built, you then go out to DOS and execute the program.

**Creating a Windows Program Using AppWizard**

AppWizard is a tool that generates an MFC project based on options that you select. AppWizard creates all the source files required to make a skeleton project that serves as a starting point for your program. You can use AppWizard to create single-document, multiple-document, or dialog box-based applications.

AppWizard creates all the source files required to build a skeleton Windows application. It also configures a project for you and allows you to specify the project directory. Although an AppWizard project is a skeleton of a future project, it uses the MFC class library to include the following functions:

• Automatic support for the common Windows dialog boxes, including Print, File Open, and File Save As

• Dockable toolbars

• A status bar

• Optional MAPI, ODBC, and OLE support

11

After answering a few questions using AppWizard, you can compile and run the first version of your application in a few minutes.

**Building Windows Applications with AppWizard**

In general, the following steps are used to build a program using AppWizard:

1. Create a program skeleton using AppWizard.

2. Create any additional resources used by the program.

3. Add any additional classes and message-handling functions using ClassWizard.

4. Add the functionality required by your program. You actually have to write some code yourself for this part.

5. Compile and test your program, using the Visual C++ integrated debugger if needed.

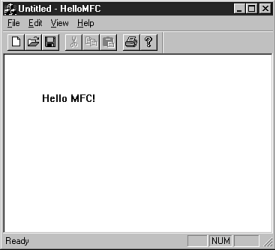
To start AppWizard and create your first Windows program, follow these steps:

1. Select New from the File menu. The New dialog box is displayed.

2. 2. Select the Projects tab. A list of project types will be displayed.

3. 3. To create an MFC-based project, select MFC AppWizard(exe) as the project type.

4. 4. Specify HelloMFC as the project name in the Project name box; a default location for your project will automatically be entered in the Location box (see Figure 1.6).



**Figure 1.6.** The New Projects dialog box for the HelloMFC project.

5. Make sure the Create New Workspace radio button is selected, and click OK to create the project.

12

6. The first MFC AppWizard screen asks for a project type, as shown in Figure 1.7. MFC AppWizard works similarly to the Developer Studio Setup Wizard, enabling you to move forward and backward using the Next and Back buttons. Select the radio button labeled Single Document and then click the Next button.

**Figure 1.7.** The first AppWizard screen for HelloMFC.

7. Move through all six MFC AppWizard screens. Each screen enables you to change a different option about the HelloMFC project. Although this example won't use any optional features, feel free to experiment with the options offered by MFC AppWizard.

8. 8. The last MFC AppWizard screen presents a list of classes that is generated for the project. Click the button labeled Finish. MFC AppWizard displays a summary of the project, listing the classes and features you selected, as shown in Figure 1.8.

**Figure 1.8.** The New Project Information dialog box for the Hello project.

9. Click the OK button to start generating files required for the HelloMFC project.

**Exploring the HelloMFC AppWizard Project**

After you create the HelloMFC project using MFC AppWizard, the Project Workspace window opens. The Project Workspace window contains four tabs, each used to show a different view of the current project:

• The *ClassView* tab displays information about the C++ classes used in the HelloMFC project. • The *ResourceView* tab displays information about the resources, such as bitmaps and dialog boxes, used in the HelloMFC project.

• The *FileView* tab displays information about the files used for the HelloMFC project. • The final view is the *InfoView*, which is used for online help information.

**Handling Output Using MFC**

The HelloMFC project already contains a function that handles output. It's called OnDraw, and it can be found in the CHelloMFCView class. When your project is created by AppWizard, the OnDraw function really doesn't do much useful work--it's up to you to supply a version of this function that does something meaningful.

To edit the CHelloMFCView class, follow these steps:

1. Click the ClassView tab in the Project Workspace window. A list of the classes used in the HelloMFC application will be displayed. Note that all the class names begin with the letter C. This is

13

a Microsoft naming convention--all of Microsoft's classes begin with C.

2. Expand the CHelloMFCView node of the tree control. A list of functions that are used in the CHelloMFCView class will be displayed.

3. Double-click the function named OnDraw. The editor will open to the OnDraw member function. Edit the CHelloMFCView::OnDraw function so that it looks like the function in Listing 1.2. You will need to remove a comment and two existing lines of code that were in the function already.

**TYPE: Listing 1.2. The OnDraw function used for HelloMFC.**

void CHelloMFCView::OnDraw(CDC\* pDC)

{

pDC->TextOut(50,50,"Hello MFC!", 10);

}

Compile the HelloMFC project by selecting Build|Build HelloMFC.exe from the main menu (or press F7). The build window displays the progress of the build, which should look something like the following:

Compiling resources...

Compiling...

StdAfx.cpp

Compiling...

HelloMFCDoc.cpp

HelloMFC.cpp

MainFrm.cpp

HelloMFCView.cpp

Generating Code...

Linking...

HelloMFC.exe - 0 error(s), 0 warning(s)

Congratulations; you have created a simple Windows program! To execute the HelloMFC project, select Execute from the Build menu or press F5 on the keyboard. The most common way to launch a project from Developer Studio is to use the debugger. To start the debugger, click the Go button on the toolbar or press F5 on the keyboard.

Figure 1.9 shows an example of the HelloMFC application running under Windows 95. **Figure 1.9.** *The HelloMFC program.*

14

One unusual aspect of the HelloMFC application is that the message is in a fixed location. If the window is resized, the text doesn't move. This is because the call to DrawText needs a fixed location for the message string in the first two parameters:

pDC->TextOut(50,50,"Hello MFC!", 10);

The third parameter is the actual message to be displayed, and the last parameter is the number of characters in the message.

In the next hour, you will learn how to display the message in the center of the main window.

**Summary**

In this chapter, you were introduced to Developer Studio and Visual C++, as well as the main tools and wizards included in Developer Studio and the MFC class library.

You also created two small programs using Visual C++: a console-mode application that displayed "Hello World!" and a Windows application that was built with AppWizard.

**Q&A**

**Q. If I know C, how much effort is needed to learn C++?**

**A.** C++ is very close to C in a number of ways. Almost every legal C program is also a legal C++ program. C++ introduces the idea of classes, which are discussed in Hour 3. A C++ compiler also has a different standard library than a C compiler. As you will see, Visual C++ makes it very easy to develop Windows programs using C++, even if you have no experience in C or C++.

**Q. Can I replace the Developer Studio editor with my own favorite editor?**

**A.** No, but you can use your favorite editor to edit files, then use Developer Studio to build those files into a final executable. You will lose many of the integrated benefits of the integrated editor if you do this, however. You can change the Developer Studio editor to emulate Brief and Epsilon editors if you prefer their keyboard mappings.

**Workshop**

The Workshop is designed to help you anticipate possible questions, review what you've learned, and begin thinking ahead to putting your knowledge into practice. The answers to the quiz are in Appendix B, "Quiz Answers."

15

**Quiz**

1. What is a library?

2. How do you build a project using Developer Studio?

3. What is a wizard?

4. What are the three most commonly used wizards?

5. How do you invoke context-sensitive help inside the editor?

6. What are the four tab views inside the Project Workspace window?

7. What MFC function is used to display output?

8. What keyboard function is used to start the build process in Developer Studio? 9. What keyboard editor command is used for Undo?

10. What is the difference between Undo and Redo?

**Exercises**

1. Change the Hello World console-mode program to display your name.

2. The first two parameters in the TextOut function call are the position coordinates for the text message. Experiment with the HelloMFC application, and change the position of the output message.

16

**- Hour 2 -**

**Writing Simple C++ Programs**

In the previous hour, you compiled some simple programs. Now it's time to learn some more details about how C++ programs work. Even simple C++ programs demonstrate basic concepts that are shared by all applications.

In this hour, you will learn

• The common elements of a C++ program

• Standard input and output in a C++ program

• The C++ preprocessor

In this hour you will build a simple C++ program that accepts input from the user and echoes it back on the screen.

**The Common Elements of a C++ Program**

Computer programs are composed of instructions and data. Instructions tell the computer to do things, such as to add and subtract. Data is what the computer operates on, such as the numbers that are added and subtracted. In mature programs, the instructions don't change as the program executes (at least they're not supposed to). Data, on the other hand, can and usually does change or vary as the program executes. A variable is nothing more than the name used to point to a piece of this data.

**Fundamental C++ Data Types**

The C++ language offers several fundamental data types. As in most other programming languages, these built-in types are used to store and calculate data used in your program. In later chapters, you use these fundamental types as a starting point for your own more complex data types.

C++ has a strong type system, which is used to make sure that your data variables are used consistently and correctly. This makes it easy for the compiler to detect errors in your program when it is compiled rather than when it is executing. Before a variable is used in C++, it must first be declared and defined as follows:

int myAge;

17

This line declares and defines a variable named myAge as an integer. A declaration introduces the name myAge to the compiler and attaches a specific meaning to it. A definition like this also instructs the compiler to allocate memory and create the variable or other object.

When the Visual C++ compiler reads the myAge definition, it will do the following:

• Set aside enough memory storage for an integer and use the name myAge to refer to it • Reserve the name myAge so that it isn't used by another variable

• Ensure that whenever myAge is used, it is used in a way that is consistent with the way an integer should be used

**Time Saver:** It's possible to define several variables on a single line, although as a style issue, many people prefer to declare one variable per line. If you want to make your source file more compact, you can separate your variables by a comma, as follows:

int myAge, yourAge, maximumAge;

This line defines three integer variables. Declaring all three variables on one line of code doesn't make your code execute any faster, but it can sometimes help make your source code more readable.

**Understanding Type Safety**

**New Term:** Some languages enable you to use variables without declaring them. This often leads to problems that are difficult to trace or fix. When using C++, you must declare all variables before they are used. This enables the compiler to catch most of the common errors in your software program. This capability to catch errors when your program is compiled is sometimes referred to as *type safety*.

You can think of type safety as a warranty that the compiler helps to enforce in your C++ program. For example, if you try to use an int when another type is expected, the compiler either complains or converts the variable into the expected type. If no conversion is possible, the compiler generates an error and you have to correct the problem before the program can be compiled.

For example, character values are normally between 0 and 127 and are stored in variables of type char. In Visual C++, a char is a single byte variable and is quite capable of storing all character values. If the compiler detects that you are attempting to store a number larger than 127 in a char, it will complain about it and issue a warning message. Listing 2.1 is an example of a program that tries to store a value that is too large in a char.

18

**TYPE: Listing 2.1. An example of a problem that can be caught by the compiler.**

#include <iostream>

using namespace std;

// This program will generate a compiler warning int main()

{

char distance = 765;

cout << "The distance is " << distance << endl; return 0;

}

To see an example of a type mismatch that is caught by the compiler, create a console mode project with Listing 2.1 as the only source file, following the steps used in Hour 1, "Introducing Visual C++ 5." The compiler flags line 6 with a warning; however, it still generates an executable program.

In order to get the program to compile with no warnings and run as expected, you must change line 5 so that the distance variable is defined as an integer:

int distance = 765;

The new version of the source code is shown in Listing 2.2.

**TYPE: Listing 2.2. A corrected version of the previous example.**

#include <iostream>

using namespace std;

// This program will compile properly.

int main()

{

int distance = 765;

cout << "The distance is " << distance << endl; return 0;

}

**New Term:** Another common data type is the *floating-point value*, or a number with a decimal point. Floating-point values are stored in float or double variables in C++ programs. These are the only two built in (or fundamental) variable types that can store floating-point values.

19

**Using Different Variable Types**

So far, you've used int and double variables, two of the fundamental types available in C++. They're called fundamental types because they are the basic data types that are a part of the language definition. There is also a set of derived types that will be covered in the next few hours. In addition, as you saw earlier with the string class, you can define your own types that work just like the built-in types. The names of the built-in types used in C++ include the following:

• bool is a Boolean variable that can have the values true or false.

• char is a variable normally used for storing characters. In Visual C++, it can have any value from - 128 to 127. If char is declared as unsigned, its range is from 0 to 255, and no negative values are allowed.

• A short int variable, sometimes just written as short, is similar to an int, but it can contain a smaller range of values. Think of it as a lightweight version of an int that can be used if data storage is a problem. A short variable can store any scalar (whole) value between -32768 and 32767. If a short is declared as unsigned, its range is from 0 to 65535.

• int is an integer value used to store whole numbers. When using Visual C++, an int is a 32-bit value so it can store any value from -2,147,483,648 to 2,147,483,647. If an int is declared as unsigned, its range is from 0 to 4,294,967,295.

• A long int, sometimes just written as long, is a scalar variable like an int, only larger when using some compilers. In Visual C++, a long int can store the same values as an int.

• A float variable is the smallest variable type capable of storing floating-point values. It is often an approximation of the value that was originally stored. In Visual C++, a float stores up to six decimal digits.

• A double variable stores floating-point values just like a float does. However, the compiler stores the value with more precision, meaning that a more accurate value can be stored. A double can store up to 15 decimal digits.

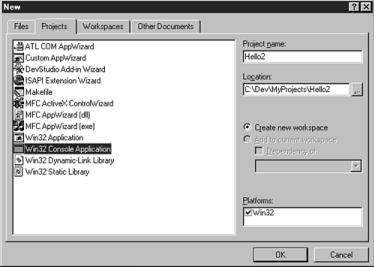
• A long double has the same characteristics as a double. However, from the compiler's point of view, they are different types. The long double type is part of the C++ language, and on some machines and compilers, the difference between double and long double is that long double has greater precision, allowing storage of more than 15 decimal digits.

**New Term:** Some of the variables in the preceding list can be declared as *unsigned*. When a variable is declared as unsigned, it can store only non-negative values. When a variable is declared as an int, it can store

20

both negative and positive numbers. However, an unsigned int can store a much larger positive value than a plain old int.

An unsigned int can store a larger positive value because the computer must use one bit of data in the memory location to handle the sign. This sign indicates whether the variable is positive or negative. Because using the sign bit reduces the number of bits that are available for storage, the maximum value for the variable is reduced by half. Figure 2.1 is an example of a variable that has been declared as int and another variable that has been declared as unsigned int.

**Figure 2.1.** Most computers can use a sign bit to determine whether a variable is positive or negative.

The fundamental variable types require different amounts of storage. As a rule of thumb, the char data type is large enough to contain all the characters in the machine's native language, or eight bits. The int type is usually the "natural" variable size for the target machine, so int variables are 32 bits in Visual C++. Table 2.1 lists the number of bytes required to store each of the fundamental types.

**Just a Minute:** Earlier versions of Visual C++ that were used with Windows 3.1 were 16-bit compilers. The natural variable size under Windows 3.1 was 16 bits, so the int type was 16 bits. The last version of Visual C++ that used 16-bit integers was Visual C++ 1.5.

21

**Table 2.1. Storage required for fundamental C++ types.**

| Type  bool  char  short  int  long  float  double  long double | Size (in bytes) |
| --- | --- |
| 1 |
| 1 |
| 2 |
| 4 |
| 4 |
| 4 |
| 8 |
| 8 |

**Variable Naming**

One important part of programming is the selection of names for your variables and other parts of your programs. The program listings you've seen so far have been very simple. As you become a more experienced user of Visual C++, you will need to establish some sort of naming convention for your identifiers.

When naming your variables, use names that are as long as necessary to indicate how the variable is used. A variable name in C++ is an example of an identifier. Identifiers in C++ are used to name variables and functions, among other things. In Visual C++, your identifiers can be literally hundreds of characters long and can include any combination of letters, numbers, and underscores, as long as the first character is a letter or underscore. Listing 2.3 is an example of several different variable declarations.

**TYPE: Listing 2.3. Some examples of good and bad variable names.**

#include <iostream>

using namespace std;

int main()

{

// Good declarations

int nEmployees; // Number of employees

char chMiddleInitial; // A middle initial

// Declarations that could be improved

int i, n, k; // What are these vars used for ? float temp; // May not be enough information char ch; // Should have more information

22

return 0;

}

No matter which technique you use to name your variables, it's important to be consistent. For example, most of the sample programs and online help examples provided as part of Visual C++ use a naming convention known as *Hungarian Notation*.

When Hungarian is used properly, it's easy to tell the logical type of variable at a glance without searching for its declaration. For example, most scalar variables such as int, long, or short are prefixed with an n. Variables that are used to store characters are prefixed with ch, as in chEntry and chInitial. Most of the sample code available from Microsoft uses Hungarian Notation, which will be used for the remainder of the code listings in this book. A listing of common Hungarian prefixes is provided in Appendix D, "Hungarian Notation."

**DO/DON'T:**

**DO** use meaningful names for your variables.

**DO** be consistent in your naming conventions.

**DO** use variable types that match your data.

**DON'T** depend on capitalization to differentiate between variables.

**Assigning Values to Variables**

In assigning values to variables, the assignment operator is just an equals sign used as follows: nFoo = 42;

This line assigns the integer value 42 to nFoo.

If a floating-point decimal value is assigned, it's assumed by the compiler to be a double, as follows: dFoo = 42.4242;

You can assign to a variable of type char in two ways. If you are actually storing a character value, you can assign the letter using single quotes as shown here:

chInitial = `Z';

The compiler converts the letter value into an ASCII value and stores it in the char variable. Small integer values can also be stored in a char, and the assignment is done just like an int variable.

chReallyAnInt = 47;

23

**Time Saver:** The char variable type is sometimes used to store small integer values. This is useful if you are storing a large number of values, because an int takes up four times the storage of a char.

**A Simple C++ Program**

In Hour 1, you created a C++ project named Hello that displayed a simple "Hello World!" message. This hour you will make a simple modification to the Hello project--the Hello2 project will ask you for a name and then use the name in the greeting. Building this project will help demonstrate some common elements found in C++ programs.

**Creating the Hello2 Project**

The first step in writing any Visual C++ program is to create a project, as you did in the first hour. To review, these are the steps required to create a console-mode project:

1. Begin by selecting File | New from the Visual C++ main menu. This will display the New dialog box.

2. Select the Projects tab in the New dialog box. A list box containing different types of projects will be displayed.

3. Select the icon labeled Win32 Console Application, as shown in Figure 2.2. You must also provide a name for the project--a default location will be provided for you automatically.

**Figure 2.2.** *The New Projects dialog box.*

After you have selected the project type and the subdirectory, click OK to create the project.

**Creating the Source File for Your Program**

The source file for the Hello2 project is shown in Listing 2.4. Unlike your first Hello program, this version collects input from the user and then outputs a greeting.

24

**TYPE: Listing 2.4. A console mode program that accepts input.**

#include <iostream>

#include <string>

using namespace std;

// Prompt the user to enter a name, collect the name, // and display a message to the user that contains // the name.

int main()

{

string userName;

cout << "What is your name? :";

cin >> userName;

cout << "Hello " << userName << "!" << endl;

return 0;

}

Open a new C++ source file and type the code shown in Listing 2.4. Remember that C++ is case-sensitive. Save the file as Hello2.cpp in the project's directory. To review, these are the steps required to open a new C++ source file and add it to the project:

1. Select File | New from the main menu, and select the Files tab in the New dialog box. 2. Select the icon labeled C++ Source File.

3. Check the Add to Project check box, and enter Hello2.cpp as the filename. 4. Click OK to close the dialog box and open the file for editing.

Compile the Hello2 project by selecting Build | Build Hello2.exe from the main menu (or press F7). If the source code was entered correctly, the project will be built with no errors, and the last line in the status window will read

Hello2.exe - 0 error(s), 0 warning(s)

If there are errors or warnings, check the source code for typographical errors and build again.

**Running the Hello2 Program**

Open a DOS window and change to the DEBUG subdirectory under the Hello2 project directory. Run the Hello2 program by typing Hello2 at the DOS prompt. The program produces the following output:

25

What is your name? :Alex

Hello Alex!

The Hello2 program accepts any name as input and uses that name for its Hello World message.

**Analyzing the Hello2 Program**

Let's take a look at the Hello2 program because it has a lot in common with much larger C++ programs. Even though it is fairly short, it has many of the elements that you will see in more complicated Windows programs later in this book.

**Include Statements**

The first line of Hello2.cpp is a message to the compiler to include another file when compiling Hello2.cpp:

#include <iostream>

This #include statement tells the compiler to look for the file named iostream and insert it into your source file. Actually, the #include statement is read by the preprocessor, a part of the compiler that scans the source file before the file is compiled.

**New Term:** Statements read by the preprocessor are known as *preprocessor directives* because they aren't actually used by the compiler. Preprocessor directives always begin with a #. You will learn more about preprocessor statements throughout the rest of the book.

**New Term:** The file iostream is an example of a *header* file. A header file contains declarations or other code used to compile your program. In order to perform common input and output operations, you must #include the iostream file.

**Just a Minute:** Traditionally, C++ header files have an .h or .hpp file extension; the standard C++ library includes files such as iostream that have no extension. For backward compatibility, the Visual C++ compiler includes older versions of the include files that have the .h extension.

The #include preprocessor directive is seen in two basic forms:

26

• When including library files, the file to be included is surrounded by angled brackets, as shown in the Hello2.cpp file shown earlier. The preprocessor searches a predefined path for the file. • When including header files that are specific to a specific application, the filename is surrounded by quotes, such as #include "stdafx.h". The preprocessor will search for the file in the current source file directory. If the file is not found, the search will continue along the predefined include path.

The second line of Hello2.cpp is also an #include directive:

#include <string>

The string header file is part of the standard C++ library. Including the string header file enables a C++ source file to use the standard string class, which simplifies using text strings in a C++ application.

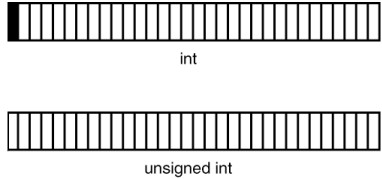
**The std Namespace**

**New Term:** A collection of names and other identifiers in C++ is known as a *namespace*. By default, any name that is introduced in a C++ program is in the *global namespace*. All names found in the standard C++ library are located in the std namespace.

Namespaces make it easier to manage names in large C++ projects, especially when using libraries or code developed by different groups of people. Before namespaces were introduced to C++, it wasn't unusual to have two or more libraries that were incompatible with each other simply because they used conflicting names.

Namespaces allow libraries to place their names into a compartment that itself has a name. As shown in Figure 2.3, two namespaces can each use a common name, in this case string; because each namespace provides a compartment for the name string, the two names do not conflict with each other.

27

**Figure 2.3.** Namespaces provide separate compartments for names used in a C++ program.

When using a name from a namespace, the namespace must be prefixed, like std::string or codev::string. Alternatively, a using namespace directive can be used to tell the compiler that an identifier can be found in the global namespace, as in the next line of the program, which tells the compiler that the names found in the program can be found in the std namespace:

using namespace std;

**Using Comments to Document Your Code**

**New Term:** A *comment* is a note provided to the person reading the source code. It has no meaning to the compiler or computer.

The next line begins with //, which is used to mark the beginning of a single-line comment in a C++ program. By default, comments are colored green by the Developer Studio editor. In contrast, int and return are colored blue to indicate that they are C++ keywords.

**Time Saver:** It's a good idea to use comments to document your code. After time has passed, you can use your comments to help explain how your code was intended to work.

**The main Function**

The next line of Hello2.cpp is the beginning of the main function.

int main()

28

The first line inside the main function is a variable declaration.

string userName;

Don't worry too much about what this means--for now, it's enough to know that userName is a string variable. A string is not one of the fundamental data types; instead, it's part of the standard library. The string type enables you to use strings of text as though they are built-in fundamental types.

Following the declaration of userName is a statement that displays a message to the user as a prompt for the user's name:

cout << "What is your name? :";

This particular statement in Hello2.cpp displays a line of characters to the console window by using the iostream object cout. The iostream library is included with every C++ compiler, although it is not technically part of the C++ language definition; instead, it's part of the standard C++ library. Performing standard input and output for your console mode program is easy using the iostream library.

The iostream library uses the << symbol for output and the >> for input to and from IO streams. Think of a stream as a sequence of bytes, like a disk file, or the output to a printer or a character-mode screen.

**Just a Minute:** One simple rule of thumb is that when you see the << symbol, the value to the right of the symbol will be output to the IO object on the left. When you see the >> symbol, data from the IO object on the left is stored in a variable to the right.

The next line of Hello2.cpp accepts input from the user and stores it in userName: cin >> userName;

The variable userName now contains whatever value was entered by the user.

The next line displays the Hello greeting and adds the contents of the userName variable. When using cout, several different components can be output one after another by separating them with the << symbol:

cout << "Hello " << userName << "!" << endl;

The last line of the main function is a return statement. When a return statement is executed, the function *returns* or stops executing, and the caller of the function is passed the value provided after the return

29

keyword. Because this return statement is inside main, the value 0 is passed back to the operating system. The return keyword can appear almost anywhere in a function. However, as a matter of style, most people prefer to have a single return statement in a function if possible.

**Summary**

In this hour, you have learned more details about C++ programs. You wrote a simple console-mode program and analyzed its parts. You also learned about the C++ preprocessor, type-safety, and variables.

**Q&A**

**Q. When I compile the Hello2 project and enter my first and last name, only the first name is displayed. How can I display my first and last names?**

**A.** When using cin to gather input as shown in the Hello2 project, white space such as the space between your first and last name will cause your names be parsed into two separate variables. You can use cin with multiple variables much like you use cout with multiple variables; just separate the variables with the >> operator. A new version of Hello2 that displays first and last names looks like this:

#include <iostream>

#include <string>

using namespace std;

int main()

{

string strFirstName;

string strLastName;

cout << "Please enter your first and last name:"; cin >> strFirstName >> strLastName;

cout << "Hello " << strFirstName << strLastName << endl;

return 0;

}

**Q. When I declare a variable, sometimes I get strange error messages from the compiler in the Build window. This is the line that causes the error:**

int my age;

**A.** In C++, all variables must be a single identifier. The compiler complains because after using

30

the identifier as a variable name, it can't figure out what to do with the identifier name. One coding style is to separate the words that make up a variable name with an underscore, like this:

int my\_age;

**Workshop**

The Workshop is designed to help you anticipate possible questions, review what you've learned, and begin thinking ahead to putting your knowledge into practice. The answers to the quiz are in Appendix B, "Quiz Answers."

**Quiz**

1. What is the difference between the cout and cin iostream objects?

2. What are the two forms of the #include preprocessor directive?

3. What type of variable is used to store character values?

4. What is the purpose of a C++ namespace?

5. How can you declare more than one variable on a single line?

6. What is type-safety?

7. What types of variable are used to store floating-point values?

8. How do you assign a value to a variable?

9. What type of variable is normally used to store integer values?

10. Why would you declare a variable as unsigned?

**Exercise**

1. Modify the Hello2 program to ask for your age in addition to your name; display the name and age in the Hello message.

31

**- Hour 3 -**

**Structures, Classes, and the MFC Class Library**

In the first two hours, you have learned some of the basic concepts behind C++, and you have written some simple programs. In this hour, you will be introduced to some more advanced Visual C++ programming topics. In particular, you will learn

• How functions are used to provide small reusable chunks of code

• How structures and classes are used to create source code and data components • How expressions and statements are used in C++ programs

• How to use the MFC class library to write Windows programs without using ClassWizard You will also build sample programs that illustrate the topics you learn about in this hour.

**Using Functions**

**New Term:** A *function* is a group of computer instructions that performs a well-defined task inside a computer program.

Functions are one of the primary building blocks of C and C++ applications. Functions provide a way to break up a large program into more manageable parts. At the same time, functions make it possible to perform the same task at various points within the program without repeating the code.

For example, If you buy a wagon, you'll find that it comes with a full set of assembly instructions and has four identical wheels. Why should the instructions repeat the steps to assemble a wheel four times? It is much easier to describe the wheel assembly process once and indicate that you perform the process for each wheel. The wheel assembly instructions are a module (function), within the full set of assembly instructions (program), that is executed four times.

Every C++ program has at least one function; this function is called main. The main function is called by the operating system when your application starts; when main has finished executing, your program has finished.

32

**Declaring Function Prototypes**

Before you can use a function, you must declare it by supplying a function prototype to the compiler. To declare a function, you specify the function's name, its return value, and a list of any parameters that are passed to it, as shown here:

int CalculateAge(int nYearBorn);

This line is a function prototype for the CalculateAge function, which takes a single integer as a parameter and returns an integer as its result. A function that returns no value is declared as returning the void type.

**New Term:** The traditional way to provide function prototypes is to place them in *header* files, which are usually named with an .h extension.

Header files that are part of the C++ standard library do not use the .h extension; two examples of standard header files are iostream and math. These header files contain all the prototypes and other declarations needed for IO streams and math functions to be compiled correctly.

**Defining Functions**

A function is defined the same way the main function is defined. All function definitions follow the same pattern; it's basically the function prototype with the function's body added to it. The function definition always consists of the following:

• The function's return value

• The function's name

• The function's parameter list

• The actual function body, enclosed in curly braces

Listing 3.1 shows how to use a function to display the Hello World! message. To run this project, create a new console-mode project named HelloFunc, using the steps described for the Hello and Hello2 projects in the first two hours.

**TYPE: Listing 3.1. The Hello World! program rewritten to use a function.**

#include <iostream>

using namespace std;

// Function prototype

void DisplayAge(int nAge);

33

int main()

{

DisplayAge(42);

return 0;

}

void DisplayAge(int nAge)

{

cout << "Hello World! I'm " << nAge << " years old." << endl;

}

Because the function doesn't return a value to the calling function, the return type is defined as void.

**Calling Functions**

In the C++ language, the act of transferring control to a function is known as *calling* the function. When a function is called, you supply a function name and a list of parameters, if any. The following steps take place when a function is called:

1. The compiler makes a note of the location from which the function was called and makes a copy of the parameter list, if any.

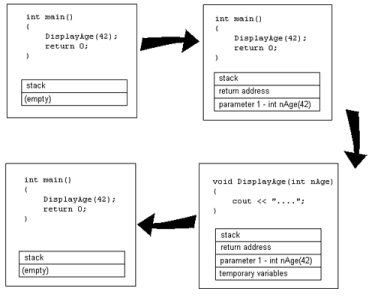
2. Any storage required for the function to execute is temporarily created.

3. The called function starts executing, using copies of the data that was supplied in the parameter list.

4. After the function has finished executing, control is returned to the calling function, and memory used by the function is released.

These steps are shown in Figure 3.1, which uses the function from Listing 3.1 as an example.

34

**Figure 3.1.** Steps involved in calling a function.

**Just a Minute:** The requirement that you declare functions before using them is an extension of the C++ type system. Because function prototypes are required, the compiler can detect errors such as incorrect parameters used in a function call.

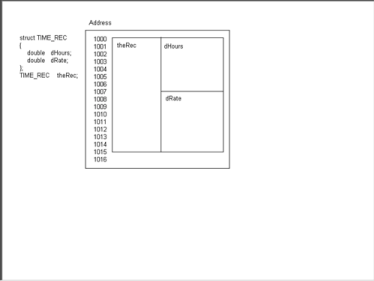
**What Are Structures?**

**New Term:** A *structure* is a data type that is an aggregate; that is, it contains other data types, which are grouped together into a single user-defined type.

**Just a Minute:** Structures are commonly used when it makes sense to associate two or more data variables.

An example of a structure is a payroll record, where the number of hours worked and the pay rate are combined in a structure, as shown in Figure 3.2.

35

**Figure 3.2.** Structures are made up of member variables.

Declaring a structure introduces a new type of variable into your program. Variables of this new type can be defined just like int, char, or float variables are defined. Listing 3.2 is an example of how a structure is typically used.

**TYPE: Listing 3.2. Using a structure to calculate a weekly salary.**

#include <iostream.h>

struct TIME\_REC

{

double dHours;

double dRate;

};

int main()

{

TIME\_REC payrollRecord;

payrollRecord.dHours = 40.0;

payrollRecord.dRate = 3.75;

36

cout << "This week's payroll information:"

<< endl;

cout << "Hours worked : " << payrollRecord.dHours << endl;

cout << "Rate :$" << payrollRecord.dRate << endl;

double dSalary =

payrollRecord.dRate \* payrollRecord.dHours;

cout << "Salary :$" << dSalary

<< endl;

return 0;

}

**What Are Classes?**

**New Term:** A *class* allows data and functions to be bundled together and used as if they are a single element. Classes typically model real-world concepts that have both data and some sort of behavior, although this is not a hard and fast rule.

Classes are similar to structures; in fact, classes really are just structures with a different name. Classes have one feature that makes them very useful for object-oriented programming: Unless a member of a class is specifically declared as public, that member is generally not accessible from outside the class. This means that you can hide the implementation of methods behind the external interface.

**Just a Minute:** Like functions, classes are an important part of the C++ programming language. In fact, one of the earliest names for C++ was C with Classes.

**New Term:** An *instance* of a class, sometimes called an *object*, is an occurrence of a class. An instance of one of your classes can be used or manipulated inside your programs.

You normally use classes to model objects in your program. Member functions, described in the next section, are used to control the state of an object, as well as to access any data contained in it.

In programs written with MFC, classes are used to model different parts of the application, such as the window frame, menus, buttons, and other controls. Member functions are used to handle specific work that needs to be handled by the class.

37

**Classes Versus Instances**

Classes and instances of classes are not the same things--this can sometimes be a confusing concept if you are new to C++ or object-oriented programming. Think of a class as the description of an object; an instance of a class is a concrete occurrence of that class.

**Constructors**

**New Term:** A *constructor*, sometimes called a "ctor," is a special member function that is created when an object of the class is created.

A constructor always has the same name as the class and never has a return value, not even void. The purpose of the constructor is to place a newly created object into a known state. Typically, constructors can allocate system resources, clear or set variables, or perform some other type of initialization.

**Destructors**

**New Term:** A *destructor*, sometimes called a "dtor," is a special member function that is called as an object is destroyed. The destructor is declared as having no return type and is never declared with a parameter list. The name of the destructor is the class name prefixed by a tilde (~) character.

It is not necessary to define a destructor unless there are specific tasks that must be performed to clean up after an object, such as releasing system resources that might have been allocated.

**Using MFC for Windows Programming**

In the first hour, you created an MFC program using AppWizard. When you use AppWizard to create a project, it might seem that you get a great deal of functionality for free. However, a great deal of code is generated--even a simple program like HelloMFC results in a large number of source files.

MFC doesn't need to be that complicated. In fact, you can write a very simple MFC program that fits in a single source file and is about one page long.

**The HelloWin MFC Example**

Listing 3.3 is an example of a simple MFC program that displays a Hello World message in the center of the client window, much like the HelloMFC program you created in the first hour.

38

**TYPE: Listing 3.3. A simple Windows program written using C++ and MFC.**

#include <afxwin.h>

// The CHelloApp class

class CHelloApp : public CWinApp

{

public:

BOOL InitInstance();

};

// The CHelloWnd class

class CHelloWnd : public CFrameWnd

{

public:

CHelloWnd();

protected:

afx\_msg void OnPaint();

DECLARE\_MESSAGE\_MAP()

};

// InitInstance - Returns TRUE if initialization is successful.

BOOL CHelloApp::InitInstance()

{

m\_pMainWnd = new CHelloWnd;

if( m\_pMainWnd != 0 )

{

m\_pMainWnd->ShowWindow( m\_nCmdShow );

m\_pMainWnd->UpdateWindow();

return TRUE;

}

else

return FALSE;

}

// Create a message map that handles one message - WM\_PAINT

BEGIN\_MESSAGE\_MAP( CHelloWnd, CFrameWnd )

ON\_WM\_PAINT()

END\_MESSAGE\_MAP()

CHelloWnd::CHelloWnd()

{

Create( NULL, "Hello" );

}

39

// OnPaint - Handles the WM\_PAINT message from Windows. void CHelloWnd::OnPaint()

{

CPaintDC dc(this);

dc.TextOut(50, 50, "Hello World!", 12);

}

// Create a single instance of the application. CHelloApp theApplication;

The simple Windows program provided in Listing 3.3 might seem large, but it's actually about half the size of a similar program written in C. Using the MFC class library enables you to use a large amount of source code that has already been written for you. There is a lot of strange-looking code in Listing 3.3, so don't try to understand it all right now.

**Building the HelloWin Example**

To build the program, create an MFC Windows project named HelloWin. Begin by selecting File | New from the Visual C++ main menu; select the Projects tab in the New dialog box. Next, select Win32 Application as the project type. You must also specify a name and location for your project, just as you did for the projects in the first two hours.

After the project has been created, open a new C++ source file document and enter the contents of Listing 3.3 exactly as they are shown. Save the file as HelloWin.cpp and add it to the project. (If necessary, refer to Hour 1, "Introducing Visual C++ 5," for specific instructions.)

Set the linking options for the project by selecting Project | Settings from the main menu. On the tab marked General is an item labeled Microsoft Foundation Classes. It will have the value Not Using MFC. Change the selection to Use MFC in a Shared Dll. You can do this by clicking on the down arrow beside the Not Using MFC selection. This opens a box where you can then make the appropriate selection.

Compile the HelloWin project by selecting Build | Build HelloWin.exe from the main menu (or Press F7).

To start the HelloWin program, select Build | Start Debug | Go from the main menu (or Press F5). Figure 3.3 shows an example of HelloWin running.

40



**Figure 3.3.** The HelloWin program displaying its message in a window.

**The Common Elements of a Windows Program**

Two elements are found in almost every Windows program; each of these elements can be found in the HelloWin program that you just compiled and ran:

• Windows are used for visible parts of an application

• Messages are used to control the interaction between an application and the Windows operating system

**Windows Are Everywhere**

One of the fundamental concepts in Windows programming is that everything you see is a window. Some examples of windows are

• Controls such as pushbuttons, list boxes, and text edit controls

• Dialog boxes and property pages

• Toolbars and menu bars

• The Windows 95 taskbar

• The DOS command box that is used for console-mode applications

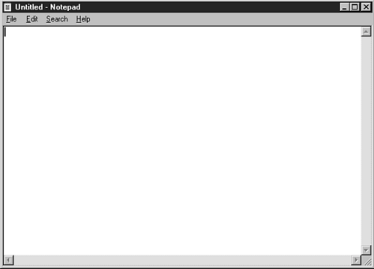
All windows have a common set of operations that can be applied to them. They are all re-sized, moved, enabled, disabled, hidden, and displayed in the same way.

**The Client and Non-Client Areas**

A window is divided into two main areas, as shown in Figure 3.4:

41

• The non-client area, which contains the border, menus, and caption area for the window • The client area, which is the area that is left over, also known as the "main" part of the window

**Figure 3.4.** Client and non-client areas of a window.

The non-client area of a window is normally maintained by Windows; your applications will normally be concerned only with the client area.

**Messages and Functions**

When Windows needs to communicate with an application, it sends it a message. A message is similar to a function call--in fact, the MFC library will route most messages as function calls into your application. For example, in an AppWizard application, the MFC library calls the OnDraw function whenever Windows sends a WM\_PAINT message.

When your application communicates with a window, it will usually send it a message. To enable or disable a control, you must send the control a WM\_ENABLE message. When using C, this process is very tedious and error prone. MFC simplifies things by providing functions that you can call and then handling the message sending for you.

42

**What Are Statements and Expressions?**

Statements and expressions are the elements defined by the C++ language that are converted into machine code by the compiler to build your C++ programs. Seems like a textbook-type definition, doesn't it? In reality, though, it is very hard to define exactly what they are. When talking about a building, we can say that it is made of bricks, boards, and other things; we can define the brick or board very easily. In the case of the C++ programming language, it is much more difficult. Here we are dealing with abstract concepts. The difference between a statement and expression is very subtle, as you will soon see. Although it appears to be confusing at first, the language will become understandable with practice. Eventually the C++ language will become as natural to you as your native language.

Just like the simple Hello programs, all C++ programs are made up of statements and expressions. Expressions and statements range from the simple statements that were shown in the Hello programs to very complex expressions that stretch across several lines.

**Statements**

All statements end with semicolons. In fact, the simplest statement is called the null statement, and it consists of only a single semicolon, as follows:

;

The null statement isn't used often; it's used only in situations in which the C++ syntax requires a statement, but no real work needs to be done.

You use a statement to tell the compiler to perform some type of specific action. For example, you know from the console mode programs you created that the following statement will cause the characters Hello World! to be displayed on your screen:

cout << "Hello World!" << endl;

**Declarations**

A declaration is another type of statement. As discussed earlier, declarations introduce a variable to the compiler. The following line is an example of a simple declaration:

int myAge;

This tells the compiler that myAge is an integer.

43

**Assignment**

An assignment expression is used to assign a value to a variable, using the assignment operator, =, as follows:

int myAge;

myAge = 135;

Every expression has a value. The value of an assignment expression is the value of the assignment. This means that the following statement assigns the value 42 to the variables yourAge and myAge:

myAge = yourAge = 42;

The program in Listing 3.4 demonstrates how to assign a value to a variable.

**TYPE: Listing 3.4. A C++ program that assigns a value to a variable.**

#include <iostream>

using namespace std;

int main()

{

int myAge;

myAge = 42;

cout << "Hello" << endl;

cout << "My age is " << myAge << endl;

return 0;

}

The assignment operator is just one example of the operators available in C++. More operators are discussed in the next section.

**Other Common Expressions and Operators**

The C++ language contains operators that you can use to write addition, subtraction, multiplication, and other expressions. Some common math operators are shown in Table 3.1.

**Table 3.1. Some common math operators used in C++.**

44

| **Operator** +  -  /  \* | **Description** |
| --- | --- |
| Addition |
| Subtraction |
| Division |
| Multiplication |

All math operators group from left to right. The multiplication and division operators have a higher precedence than the addition and subtraction operators. This means that the following expressions are equivalent: a + 5 \* 3 a + 15 You can use parentheses to force an expression to be evaluated in a preferred order. Note the grouping of the following expression: (a + 5) \* 3 This expression adds 5 to the value stored in a and then multiplies that value by 3. The math operators can also be combined with an assignment operator, as follows:

int myAge;

myAge = 40 + 2;

The expression 40 + 2 has a value of 42. After that value is calculated, the value of the expression is stored in the myAge variable.

**Rectangles and Regions**

The rectangle is a fundamental component of most Windows programs. Because most windows and controls are rectangular, it isn't surprising that one of the most commonly used data structures in Windows programming is used to represent a rectangle.

Rectangles are often used to represent the position or size of all types of windows: main windows as well as controls, toolbars and dialog boxes. There are two basic types of rectangle coordinates:

• Screen rectangle coordinates, which place a rectangle in relationship to the entire screen • Client rectangle coordinates, which always have their top and left values set to zero and provide the size of a rectangle that represents a window's client area

Screen rectangle coordinates are often used when moving a window in relation to the entire screen. Client rectangles are most commonly used when positioning controls or drawing inside a control or other window.

When requesting the dimensions of a rectangle, you must pass a CRect variable to one of the Windows rectangle functions. The following two lines of code declare an instance of CRect as a variable and pass it to the GetClientRect function:

45

CRect rcClient;

GetClientRect(rcClient);

The next example uses a client area rectangle to display a message to the user, just like the HelloMFC program in the first hour. The new example will draw the message in the center of the client area; if the window is resized, the message will be redrawn in the center of the new rectangle.

Create an MFC AppWizard application named HelloRect, following the steps presented in the first hour. Modify the OnDraw function found in the CHelloRectView class so that it looks like the function shown in Listing 3.5.

**TYPE: Listing 3.5. Using a rectangle to center a message in a window.**

void CHelloRectView::OnDraw(CDC\* pDC)

{

CRect rcClient;

GetClientRect(rcClient);

pDC->DrawText("Hello Client Rectangle!",

-1,rcClient,

DT\_SINGLELINE |

DT\_CENTER |

DT\_VCENTER );

}

Build the HelloRect application, and run it from Developer Studio. Note that if you resize the window, the message is redrawn so that it remains in the center of the client area.

**Summary**

In this hour, you have learned about some of the more advanced building blocks that make up C++ programs: functions, structures, and classes. You also looked at some basic information about the MFC class library and built an MFC application without using ClassWizard.

**Q&A**

**Q. What is the difference between a rectangle that uses screen coordinates and a rectangle that uses client coordinates?**

**A.** Every window in a Windows application can be represented by a rectangle; this rectangle will typically use either screen or client coordinates. The rectangle that results from these coordinates is always the same, the difference is only in the point of reference that is used to

46

measure the rectangle.

**Q. Can a structure have member functions?**

**A.** Absolutely. A class and a structure are exactly the same, except that all members of a structure are accessible by default, while class members are private (not accessible) by default. You will learn more about access restrictions in the next hour.

**Q. Why is no function prototype required for main()?**

**A.** The short answer: because the C++ standard says you don't need one. The purpose of function prototypes is to introduce new functions to the compiler; because every C++ program is required to have a main function, no function is necessary.

**Workshop**

The Workshop is designed to help you anticipate possible questions, review what you've learned, and begin thinking ahead to putting your knowledge into practice. The answers to the quiz are in Appendix B, "Quiz Answers."

**Quiz**

1. What are some examples of the different types of windows found in a Windows application? 2. What is a function?

3. What are the four parts of a function definition?

4. How are classes different from structures?

5. What function is called when an instance of a class is created?

6. What function is called when an instance of a class is destroyed?

7. What is the difference between the client and non-client areas?

8. What is the value of the expression a = 42?

9. What symbol is used for multiplication?

10. What symbol is used for division?

**Exercises**

1. Write a console-mode program that asks for a distance in miles and converts the distance into feet. There are 5,280 feet in a mile.

2. Modify the HelloWin program to display different messages in different parts of the main window.

47

48

**- Hour 4 -**

**Dialog Boxes and C++ Classes**

In this hour, you will learn about two fundamental concepts used when creating Windows programs using C++: object-oriented design and dialog boxes. The following topics are covered in this hour:

• An introduction to object-oriented design concepts

• Using dialog boxes in your Windows applications

• Creating dialog-based projects

• Adding controls to dialog boxes

• Creating new classes using ClassWizard

Also during this hour, you will create two sample projects that demonstrate how you can use dialog boxes in your applications.

**What Is Object-Oriented Design?**

One of the design goals for the C++ language was to provide a language that supported object-oriented programming and design. *Object-oriented design* involves classifying real-world objects and actions as classes that can be created, manipulated, and destroyed.

The data that makes up an object, and the functions that are performed on that object, are combined to form a *class*, or a description of that object. Classes can inherit functionality from other objects, and you easily can add new classes that leverage existing classes.

**Just a Minute:** Object-oriented programming is not new; the first language to support object-oriented programming, Simula, has been around since the mid-1960s.

**Why Use Object-Oriented Design?**

In traditional, structured design, the data manipulated by a program and the functions that manipulate the data are separate. Reusing parts of a design built with structured design techniques often is difficult unless the new design is very similar to the old design.

49

Object-oriented design is useful because it can be used to create designs that can be reused and extended. A design, or a portion of a design, can be reused in future programs much like a hardware component, such as a computer chip, a disk drive, or a sound card. Object-oriented designs describe the object's class completely, so each class is easily reused because the data and functions described by the class are integrated.

Because you can hide the implementation of a class behind an interface, changing the implementation details of a class without affecting users of that class--as long as the interface doesn't change--is easy. For example, the Tab control was not available in versions of Windows before Windows 95. The MFC CPropertyPage class was rewritten for MFC 4.0 to take advantage of the new Tab control without impacting users of that class, except that the class is now more efficient.

**Describing Objects in a Class**

**New Term:** In C++, objects are described by a *class*. A class is just a description of the object that can be created and the actions that can be performed on it.

A C++ class has two main parts:

• The *class declaration*. This contains the class interface and information about data members for the class. The class interface usually is located in a header file having a .H suffix. Any file in your program that uses the class must use the #include directive so that the class declaration is added to the source file by the preprocessor.

• The *class implementation*. This includes all the member functions that have been declared as part of the class. The class implementation usually is located in a file that has a .CPP suffix.

**What Is a Dialog Box?**

**New Term:** A *dialog box* is a specialized window that is used to provide feedback or collect input from the user. Dialog boxes come in all shapes and sizes, ranging from simple message boxes that display single lines of text to large dialog boxes that contain sophisticated controls.

**New Term:** The most commonly used type of dialog box is a *modal dialog box*. A modal dialog box prevents the user from performing any other activity with the program until the dialog box is dismissed.

Dialog boxes are also used for one-way communication with a user, such as "splash screens" used to display copyright and startup information as a program is launched. The opening screen displayed by the Visual C++ Developer Studio and Microsoft Word are two examples of dialog boxes used for one-way communication. Dialog boxes are sometimes used to notify the user about the progress of a lengthy operation.

50

**Just a Minute:** Dialog boxes provide a convenient way for users to interact with Windows programs. Users expect most interaction with a Windows program to take place through dialog boxes. All dialog boxes have certain things in common; these common characteristics make the user's life easier, because users don't need to learn and relearn how dialog boxes work from program to program.

There are several different types of dialog boxes, and each of them has a specific purpose. This hour covers three main types of dialog boxes:

• Message boxes

• Modal dialog boxes

• Modeless dialog boxes

**Understanding Message Boxes**

The simplest type of dialog box is the message box, which is used to display information. This type of dialog box is so simple you can call it with just one line of code using the MFC class library. For example, to display a message box using default parameters supplied by MFC, just use this line:

AfxMessageBox( "Hello World" );

This line of code creates a message box with an exclamation mark inside a yellow triangle. There are several additional options for the icon displayed in a message box, as you will see later.

**Using Dialog Boxes for Input**

When most people think of dialog boxes, they think of the dialog boxes that collect input from a user. Dialog boxes are often used to contain controls that are used to handle user input. You can include in a dialog box a wide range of controls. In fact, a major portion of this book covers the various types of controls available in Windows.

**New Term:** Some dialog boxes are needed so often in Windows programs that they have been included as part of the operating system. These dialog boxes, known as *common dialog boxes*, are available by calling a function and don't require you to create a dialog box resource. There are common dialog boxes for opening and selecting files, choosing fonts and colors, and performing find and replace operations. Many of the common dialog boxes are covered later in the book. For example, in Hour 13, "Fonts," you will use a common dialog box to select a font.

**New Term:** A dialog box that is *modeless* enables other activities to be carried out while the dialog box is still open.

51

An example of a modeless dialog box is the Find and Replace common dialog box used by Developer Studio. When the dialog box is open, you can still make selections from the main menu, and even open other dialog boxes. In contrast, all other Developer Studio dialog boxes are modal. As long as they are open, the user cannot interact with the other parts of Developer Studio.

**How Are Dialog Boxes Used?**

Developer Studio makes using dialog boxes in a Windows program easy. All the necessary steps are automated, and the tools used to create the dialog box and include it in a project are all integrated.

**Adding Message Boxes**

As discussed earlier, you can add message boxes to your program using a single line of code. You must supply at least a single parameter: the text that is displayed inside the dialog box. Optionally, you can also specify an icon style and a button arrangement pattern. The types of icons that are available for message boxes are shown in Figure 4.1.



**Figure 4.1.** Icons you can include in a message box.

Each of the icons in Figure 4.1 has a specific meaning. When most Windows programs display a message box, they use a standard icon for each message. When programs use the same icons consistently, users find it much easier to understand the meanings of information provided with message boxes. The meaning and style name for each icon is shown in Table 4.1.

**Table 4.1. Icons used in Windows message-box dialog boxes.**

| **Icon Displayed** Exclamation mark Warning An "i" in a circle Question mark Stop sign | **Meaning** | **Message Box Style** |
| --- | --- | --- |
|  | MB\_ICONEXCLAMATION |
| Information | MB\_ICONINFORMATION |
| Question | MB\_ICONQUESTION |
| Error | MB\_ICONSTOP |

52

In addition, you can specify a button arrangement to be used in the message box. By default, a single button labeled OK is included in the message box. However, sometimes it's convenient to ask a user a simple question and collect an answer. One use for these button arrangements is to ask the user what action to take during an error. For example, the following code displays a message box that contains a question mark icon and asks the user whether the current file should be deleted:

int nChoice = AfxMessageBox( "Overwrite existing file?", MB\_YESNOCANCEL |

MB\_ICONQUESTION );

if( nChoice == IDYES )

{

// Overwrite file

}

The user can choose between buttons marked Yes, No, and Cancel. Table 4.2 gives the different button arrangements possible for a message box.

**Table 4.2. Button arrangements.**

| **Message Box Style**  MB\_ABORTRETRYIGNORE MB\_OK  MB\_OKCANCEL  MB\_RETRYCANCEL  MB\_YESNO  MB\_YESNOCANCEL | **Buttons Included in Dialog Box** |
| --- | --- |
| Abort, Retry, and Ignore |
| OK |
| OK and Cancel |
| Retry and Cancel |
| Yes and No |
| Yes, No, and Cancel |

The message-box return value indicates the button selected by the user. Table 4.3 is a list of possible return values and the choice made by the user.

**Table 4.3. Message-box return values.**

| **Return Value** IDABORT IDCANCEL IDIGNORE IDNO  IDOK  IDRETRY | **Button Pressed** Abort  Cancel  Ignore  No  OK  Retry |
| --- | --- |

53

| IDYES | Yes |
| --- | --- |

**Using the Bitwise OR Operator**

In an earlier code fragment, a vertical bar was used to separate two different options for the AfxMessageBox function. The vertical bar is the *bitwise OR operator*, which is used to combine the bit patterns of two or more values.

Unlike adding two operands, the values are combined "bitwise," meaning that the two values are compared bit by bit. If a particular bit is high in either operand, the result will have that bit enabled. If a particular bit is low in both operands, the result will be zero. For example, this expression is equal to 7:

4|3

However, the result of the following is also equal to 7:

4|7

Each possible parameter has a unique bit pattern, enabling you to use the bitwise OR operator when you combine parameter values in AfxMessageBox or other MFC function calls.

When using the bitwise OR operator with AfxMessageBox, you can combine one icon style and one button style. You can't combine two icon styles or two button styles. If no styles are provided, the message box will contain the exclamation-mark icon and an OK button.

**Adding a Dialog Box**

Adding a dialog box to a program usually takes four steps:

1. Design and create a dialog box resource using the Developer Studio resource tools. 2. Use ClassWizard to create a C++ class derived from CDialog that will manage the dialog box. 3. Add functions to handle messages sent to the dialog box, if needed.

4. If the dialog box is selected from the main menu, the menu resource must be modified and message handling functions must be created using ClassWizard.

Each of these steps is covered in the following sections.

54

**Understanding Resources**

Dialog boxes are just specialized types of windows. However, because they commonly are used for short periods of time they usually are stored as program resources and loaded only when needed. You can see this behavior when running a Windows program on a machine that has little free memory. Every time a dialog box is opened, the hard disk is accessed to load the dialog box resources from the EXE.

Menus and accelerators, which are covered in Hour 10, "Menus," are two types of resources. Here are some of the other resource types used by Windows programs:

• *Bitmaps* store images such as the logo from the Visual C++ opening, or splash screen. You will learn more about bitmaps in Hour 15, "Using Bitmaps."

• *Cursors* indicate the current mouse position. A program can modify the cursor to indicate that a specific action can be taken with the mouse at its current position, or for other user-feedback purposes. You will learn more about cursors in Hour 14, "Icons and Cursors."

• *Dialog boxes* are windows used to interact with a program's user. The message box covered earlier in this hour is one example of a dialog box.

• *Icons* are small bitmaps in a special format that can be used to represent another object in a Windows program. For example, icons are often used to represent programs and documents that aren't currently visible.

In this hour you will learn how to create and use dialog box resources. Later hours deal with the other resource types.

**Creating a Dialog Box Resource Using Developer Studio**

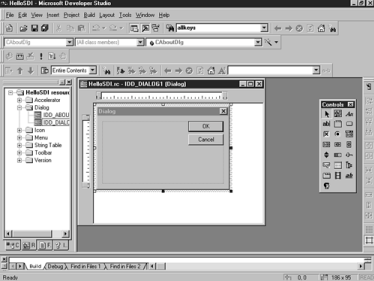
Developer Studio enables you to create a dialog box and configure it visually. You can add and size controls by using a mouse. You can set attributes for the dialog box and its controls with a click of a mouse button.

Before using the following steps, create a Single Document MFC AppWizard application named HelloSDI, following the steps presented in the first hour. Create a new dialog box resource for the HelloSDI project using either of the following methods:

• Select Resource from the Insert menu, then select Dialog as the resource type.

• Right-click the Dialog folder in the Resource tree, and select Insert Dialog from the pop-up menu. With either of these methods the dialog box editor is displayed, as shown in Figure 4.2.

55

**Figure 4.2.** The Developer Studio dialog box editor.

The dialog box that is displayed for editing initially contains two button controls, one labeled OK and another labeled Cancel. As you will learn in Hour 5, "Button Controls," these are two standard dialog box controls. The MFC class library usually handles the operation of these controls.

**Customizing the Dialog Box's Properties**

Every dialog box has properties that you can display by right-clicking and selecting Properties from the pop up menu. Here are the dialog box properties under the tab labeled General:

• ID: Normally set to something like IDD\_DIALOG1. Naming dialog boxes with an identifier that begins with IDD\_ is an MFC convention, although you should try to name your dialog boxes with a more meaningful name; rename this dialog box IDD\_HELLO.

• Caption: Normally set to a default of Dialog. You should change this to something more meaningful as well, such as Hello for the sample dialog box.

• Menu: Normally cleared because few dialog boxes use a menu.

• X Pos: Normally cleared to use the default positioning for the dialog box.

• Y Pos: Normally cleared to use the default positioning for the dialog box.

• Font Name: Contains the current font used by the dialog box.

56

• Font Size: Contains the current font size used by the dialog box.

**Time Saver:** There is also a pushbutton labeled Font... that you can use to change the default font for the dialog box. However, just because you can doesn't mean that you should. Windows enables users to set the font style used in dialog boxes; many users,

such as the visually impaired, might need specific fonts to be able to use your dialog box.

Like all windows, a dialog box has several style attributes. You can display these attributes by selecting the tab labeled Styles. Here are the default values for the following attributes:

• Style: Usually set to Popup for most dialog boxes. In the case of special dialog box templates used in form views or dialog bars, the style is set to Child.

• Border: Set to Dialog Frame for most dialog boxes.

• Minimize Box: Creates a minimize box for the dialog box. This check box is cleared for most dialog boxes, indicating that no minimize box is provided.

• Maximize Box: Used to create a maximize box for the dialog box. This check box is cleared for most dialog boxes, indicating that no maximize box is provided.

• Title Bar: Creates a title bar for the dialog box. This check box is almost always checked because most dialog boxes have a title bar.

• System Menu: Used to indicate that a system menu should be provided for the dialog box. This check box is normally checked.

• Horizontal Scroll: Used to create a scrollbar for the dialog box. This check box is almost always cleared because dialog boxes rarely use scrollbars.

• Vertical scroll: Used to create a vertical scrollbar for the dialog box. Like the horizontal scrollbar, this attribute is rarely used.

• Clip Siblings: Used only with child windows. This check box is normally cleared. • Clip Children: Used for parent windows. This check box is rarely checked for most dialog boxes.

The tab labeled More Styles contains additional properties for the dialog box:

• System Modal: Creates a system-modal dialog box. If this option is enabled, the user cannot switch to another program.

• Absolute Align: Used to indicate how the dialog box is positioned when initially displayed. If this check box is checked, the dialog box is aligned with the screen instead of with the parent window.

57

• No Idle Message: Prevents a particular window message, WM\_ENTERIDLE, from being sent when the dialog box's message queue is empty. This check box is normally cleared. • Local Edit: Used to specify how an edit control's memory is allocated. This check box is normally cleared, which means edit controls use memory outside the program's data segment. • Visible: Used to specify that the dialog box should be visible when first displayed. This check box is usually checked. In the case of form views, this check box is cleared. Form views are discussed in Hour 23, "Advanced Views."

• Disabled: Indicates that the dialog box should be disabled when initially displayed. This check box is usually cleared.

• 3D-Look: Gives the dialog box a three-dimensional appearance. This check box is usually cleared. • Set Foreground: Forces the dialog box to be placed into the foreground. This check box is usually cleared.

• No Fail Create: Tells Windows to create the dialog box even if an error occurs. This check box is usually cleared.

• Control: Creates a dialog box resource that can be used as a child control. This check box is usually cleared.

• Center: Causes the dialog box to be centered when it is initially displayed. This check box is usually cleared.

• Center Mouse: Places the mouse cursor in the center of the dialog box. This check box is usually cleared.

• Context Help: Adds a question mark icon for context-sensitive help in the title bar. This check box is usually cleared.

Advanced styles are located under the tab labeled Extended Styles. These styles are rarely used and aren't discussed in this book.

**Adding a Static Text Control**

A simple control that you can add to the dialog box is a static text control. The static text control requires no interaction with the dialog box; it is often used as a plain text label for other controls contained by the dialog box. To add a static text control, follow these steps:

1. Select the Static Text control icon on the control toolbar. The cursor changes shape to a plus sign when moved over the dialog box.

2. Center the cursor over the dialog box, and click the left mouse button. A static text control is created and contains the label Static.

3. Change the label of the static text control by right-clicking the control and selecting Properties from the shortcut menu; change the caption to "Hello World".

58

The static text control is visible whenever the dialog box is displayed. Text controls are an excellent choice for labeling controls or messages that are not likely to change. Experiment with changing the size and position of the static text control by dragging its edges with the mouse.

**Creating a Class for the Dialog Box**

You can use the CDialog class to manage most of the interaction with a dialog box in your program. The CDialog class provides member functions that make a dialog box easy to use. You should use ClassWizard to derive a class from CDialog that is specifically tailored for your dialog box.

To start ClassWizard, use any of these methods:

• Press Ctrl+W almost any time in Developer Studio.

• Select ClassWizard from the View menu.

• Right-click anywhere in the dialog box editor, and select ClassWizard from the pop-up menu.

If ClassWizard knows that a new resource has been added, such as IDD\_HELLO, a dialog box asks you to choose between two options for the new dialog box resource:

• Create a new class.

• Select an existing class.

You should almost always choose to create a new dialog box class unless you are reusing some existing code. A New Class dialog box is displayed, as shown in Figure 4.3.

Values provided to the New Class dialog box are used by ClassWizard to create a class that will manage the new dialog box resource. Use the values from Table 4.4 to fill in the values for the IDD\_HELLO dialog box.

**Table 4.4. Sample values for the New Class dialog box.**

| **Control**  Name  File Name  Base Class  Dialog ID  OLE Automation None | **Value** |
| --- | --- |
| CHelloDlg |
| HelloDlg.cpp |
| CDialog |
| IDD\_HELLO |
|  |

59



**Figure 4.3.** *The New Class dialog box.*

Click the button labeled OK. The CHelloDlg class is generated, and two files will be added to your project:

• The HelloDlg.h file contains the class declaration.

• The HelloDlg.cpp file contains the class definitions.

**Adding a Message Handler for WM\_INITDIALOG**

Dialog boxes receive the WM\_INITDIALOG message from the operating system when all the controls owned by the dialog box have been created. Most dialog boxes use the WM\_INITDIALOG message to perform any initialization that is needed.

After you have added the CHelloDlg class to the HelloSDI project, you can use ClassWizard to add a message-handling function for messages such as WM\_INITDIALOG.

To add a message handler for WM\_INITDIALOG, follow these steps:

1. Open ClassWizard by pressing Ctrl+W or by right-clicking in a source code window and selecting ClassWizard from the menu.

2. Select the tab labeled Message Maps and select from the Class Name combo box the class that will handle the message--in this case, CHelloDlg.

3. Select the object that is generating the message from the Object ID list box--in this case,

60

CHelloDlg. A list of messages sent to the dialog box will be displayed in the Messages list box.

4. Select the WM\_INITDIALOG message from the Messages list box and click the Add Function button. ClassWizard will automatically add the OnInitDialog function to the CHelloDlg class.

5. Click OK to close ClassWizard.

The CHelloDlg::OnInitDialog function doesn't really need to initialize any variables, so you can display a message box instead. Edit OnInitDialog so that it looks like the function in Listing 4.1.

**TYPE: Listing 4.1. The CHelloDlg::OnInitDialog function.**

BOOL CHelloDlg::OnInitDialog()

{

CDialog::OnInitDialog();

AfxMessageBox( "WM\_INITDIALOG received" );

return TRUE;

}

**Adding a Menu Choice for the New Dialog Box**

To add a menu item to the menu used by HelloSDI follow the steps in this section. Don't worry too much about what's going on here; you'll learn more about menus in Hour 10.

Menus are stored in your project as resources. To display the current menu resources, select the ResourceView tab in the project workspace window. Expand the resource tree to show the different resource types defined for the current project; one of the folders is labeled Menu.

Open the Menu folder to display the single menu named IDR\_MAINFRAME. Open the menu resource by double-clicking the menu resource icon. The menu is displayed in the resource editor ready for editing. Clicking any top-level menu item displays the pop-up menu associated with that item, as shown in Figure 4.4.

61

**Figure 4.4.** Using the Developer Studio resource editor to edit a menu resource.

The last item of every menu is an empty box. This box is used to add new menu items to the menu resource. All menu items are initially added to the end of a menu resource and then moved to their proper position. To add a new menu item, follow these steps:

1. Double-click the empty box on the File menu to display the Menu Properties dialog box.

2. To add a menu item, provide a menu ID and caption for the new menu item. For this example, enter ID\_FILE\_HELLO as the menu ID and &Hello as the menu caption.

3. Click anywhere outside the properties dialog box to return to the editor.

After adding a menu item, the next step is to add a message-handling function to handle the new menu item. To add a message-handling function for the ID\_FILE\_HELLO menu item, follow these steps:

1. Open ClassWizard by pressing Ctrl+W or by right-clicking in a source code window and selecting ClassWizard from the menu.

2. Select the tab labeled Message Maps and select from the Class Name combo box the class that will handle the message--in this case, CMainFrame.

3. Select the object that is generating the message from the Object ID list box--in this case, ID\_FILE\_HELLO. Two message-handling functions are displayed in the Messages list box.

4. Select the COMMAND message from the Messages list box and click the Add Function button. Accept the default name suggested by ClassWizard for the function name: OnFileHello.

5. Click OK to close ClassWizard.

62

Edit the CMainFrame::OnFileHello function so that it looks like the function provided in Listing 4.2.

**TYPE: Listing 4.2. The message-handling function for the Hello menu item.**

void CMainFrame::OnFileHello()

{

CHelloDlg dlgHello;

if( dlgHello.DoModal() == IDOK )

AfxMessageBox( "OK button pressed" );

else // IDCANCEL

AfxMessageBox( "Cancel button pressed" );

}

Add an #include statement in MainFrm.cpp that includes the class definition for CHelloDlg, found in HelloDlg.h, by adding the following line just above the include statement for MainFrm.h:

#include "HelloDlg.h"

Compile and run the HelloSDI project. When the DoModal member function is called, the IDD\_HELLO dialog box is displayed. The function call does not return until you close the dialog box by pressing one of the dialog box's buttons. If you press OK, the return value is IDOK. If you press Cancel, the return value is IDCANCEL.

**Creating Dialog Box-Based Projects**

**New Term:** A *dialog box-based project* uses a dialog box as the main window of a simple program. For example, many of the utilities found in the Windows 95 Control Panel are dialog box-based.

A dialog box-based program has a menu that is accessed through the system menu at the upper-left corner of the dialog box's caption bar. A dialog box-based project is often used to build very small programs that interact with the user through a single dialog box. The program can be much smaller and easier to program because the number of classes created by AppWizard is reduced by about half.

**Time Saver:** If your program must have sophisticated menus, it should not be dialog box based.

63

A user can easily operate a dialog box-based program. There is only one dialog box window, no menu, and all the available controls usually are initially visible. There are no hidden dialog boxes or menu items, and the user can usually see exactly which operations should be carried out.

**AppWizard Support for Dialog Box-Based Projects**

You can create a dialog box-based program using AppWizard, just like the SDI program you built earlier in this hour. Building a dialog box-based project is one of the initial options offered by AppWizard.

Because a dialog box-based project is much simpler than an SDI or MDI project, fewer steps are required when using AppWizard. Only four wizard pages are presented by AppWizard when building a dialog box based project, versus the six pages required for an SDI or MDI project.

To create a dialog box-based project using AppWizard, follow these steps:

1. Open MFC AppWizard by creating a new project workspace, as you have in previous hours. For the purpose of building an example for this hour, use the name HelloDialog.

2. When the opening screen for AppWizard appears, select a dialog box-based project as the project type.

3. Accept the default project settings suggested by AppWizard and press the Finish button. You can also browse through the Wizard pages and change the default settings. AppWizard creates the dialog box based project for you, just as it did earlier in the hour for the HelloSDI project.

**Exploring the HelloDialog AppWizard Project**

After you create the HelloDialog project, take some time to explore the project workspace. Much of the project workspace looks just as it does for an SDI project. There are four tabs for the different project workspace views, and several files and resources have been created for you.

There are several differences between a dialog-based project and an SDI or MDI project:

• No menu resource is created for the project. Because the project uses a dialog box as its main window, there's no need for a menu in most cases.

• There are no document or view classes. Dialog-based projects are intended to be very simple applications that don't require Document/View support.

• There are two dialog box resources. The main window for the project is a dialog box, as is the About box. The names of the two dialog box resources for the HelloDialog project are

IDD\_ABOUTBOX and IDD\_HELLODIALOG\_DIALOG.

64

**Using the Dialog Box Editor**

Open the dialog box editor by double-clicking the IDD\_HELLODIALOG\_DIALOG icon. The IDD\_HELLODIALOG\_DIALOG dialog box is displayed in the dialog box editor, along with a dockable control toolbar or palette. The dialog box will already contain a static text control. Modify the static text control so that it reads Hello Dialog Project, as shown in Figure 4.5.



**Figure 4.5.** The main dialog box for the HelloDialog project.

Build and run the HelloDialog project. Because it is much smaller, the HelloDialog project will compile and launch faster than an SDI or MDI project. For that reason many of the examples in this book that deal with controls will use dialog box-based projects.

**Summary**

This hour began with an introduction to object-oriented design. In this hour, you also learned about dialog boxes and how they are used in programs written for Windows. This hour also covered the support provided by Developer Studio, including ClassWizard, the MFC class library, and the dialog box editor.

**Q&A**

**Q. When I display a modal dialog box, no other part of the user interface can be used; does my application also stop functioning while the dialog box is displayed?**

**A.** A modal dialog box prevents the user from accessing other parts of your application; it does not prevent Windows from sending events to your message-handling procedures. Your application will continue to work normally while displaying a modal dialog box.

**Q. Why are C++ classes always split into two files? Wouldn't it be easier to have only a single file that defines the class as is done with Java?**

65

**A.** A key part of most languages that support object-oriented programming is the idea that the description of a class should be kept separate from its implementation. This fits in with the notion of information hiding, where unnecessary details are hidden whenever possible. In a well-designed C++ class, the implementation is considered a detail that the consumer doesn't need to be concerned with.

In Java, the class is always defined inside the class declaration, and they are never separated. This simplifies the work required for the compiler and runtime system. However, it also forces you to deal with implementation details when reading the class declaration.

If you prefer to define a class inside the class declaration, C++ supports that coding style; just include the function body after its declaration:

class CFoo

{

int m\_nBar;

public:

CFoo()

{ m\_nBar = 0; }

void SetBar(int newVal)

{ m\_nBar = newVal; }

int GetBar() const

{ return m\_nBar; }

};

**Workshop**

The Workshop is designed to help you anticipate possible questions, review what you've learned, and begin thinking ahead to putting your knowledge into practice. The answers to the quiz are in Appendix B, "Quiz Answers."

**Quiz**

1. What is the difference between a modal and modeless dialog box? 2. What message is sent to a dialog box for initialization purposes?

3. What is the file extension used for C++ class declaration files?

4. What is the file extension used for C++ class implementation files? 5. What message box style is provided by default when using AfxMessageBox?

66

6. What message box style should be used when reporting an error to a user? 7. What MFC class is used to manage dialog boxes?

8. What member function is called to pop up a modal dialog box?

9. If the user presses the Yes button in a message box, what return value is provided to AfxMessageBox?

10. If the user presses the No button in a message box, what return value is provided to AfxMessageBox?

**Exercises**

1. Change the HelloSDI example so that the message box displayed for WM\_INITIDIALOG uses the information icon.

2. Add a second static text label to the HelloDialog project's main dialog box that displays your name.

67

**- Hour 5 -**

**Button Controls**

Button controls are probably the most flexible controls available in Windows. Before learning about buttons, though, it's important to begin with a short lesson about conditional expressions in C++ programs. In this hour you will also learn about

• Using the different types of button controls provided by Windows

• Using the MFC CButton class that is used to manage button controls

• Using the MFC CWnd class to enable and disable controls

Later this hour, you will add each type of button to a dialog box-based project. You will also use ClassWizard to add button events and member variables for the dialog box's button controls.

**What Are Conditional Expressions?**

**New Term:** A *conditional expression* is an expression that results in a true or false value.

Most programs exercise some type of control over their execution flow using conditional expressions. They perform different actions based on varying conditions as the execution progresses. Then, they repeat these actions until all their tasks are complete. For example, a Windows program might need to search for a certain record from a database, or might take different actions depending on the messages that are sent to it.

**New Term:** A *selection statement* uses a conditional expression to pick a particular path of execution in your program. This is similar to choosing a fork in the road.

**New Term:** A *sequence statement* uses a conditional expression to determine how often to execute a part of your program.

**Selecting an Execution Path with Selection Statements**

The first set of control statements to look at are the selection statements. If your program must take a particular action only if a certain condition is true, or if a user must make a choice from a list of possible items, these statements are for you.

68

**Just a Minute:** All selection statements work by evaluating an expression, then taking an action based on the value of that expression.

**Using the if Statement**

The if statement enables one or more statements to be executed only if an expression inside the parentheses is true. If necessary, values inside the parentheses are converted into Boolean values, with zero being converted to false and all non-zero values converted to true.

Listing 5.1 provides a function that shows how the if statement is used. If the parameter passed to the function is greater than zero, the function returns a value of true.

**TYPE: Listing 5.1. A function that returns true if a positive number is passed to it.**

bool IsPositive( int nCheckValue )

{

bool bReturn = false;

if( nCheckValue>0)

bReturn = true;

return bReturn;

}

**Using Compound Statements**

The statement controlled by an if statement is executed only when the test condition is true. If more than one statement must be executed, group the statements together to form a compound statement. Compound statements are often called *blocks* because they group statements into blocks of code.

A compound statement begins and ends with curly braces, just like a function body. All the statements within a compound statement that follows an if statement are executed when the test condition is true, as shown in Listing 5.2.

**TYPE: Listing 5.2. Using a compound statement to group several statements together.**

void PrintTest(bool bShouldPrint)

{

69

if( bShouldPrint == true )

{

cout << "A short demonstration of" << endl;

cout << "a compound statement - also" << endl; cout << "known as a block." << endl;

}

}

In Listing 5.2, the test for equality is made using ==, the equality operator.

**CAUTION:** A common mistake is to use =, which is the assignment operator.

A standard code-formatting convention is to visually nest each conditional "level" of your source code by indenting statements, as in Listings 5.1 and 5.2. Indentation helps make your code more readable because it helps make the flow of control in your source code easy to see.

**Using else with if Statements**

You can couple an else statement with an if statement to create an either/or selection. When the expression tested by the if statement is true, the first statement (or block statement) is executed. When the expression is false, the statements grouped with the else statement are executed instead.

Listing 5.3 provides an example of a function that uses the if and else statements. This function always returns the larger of two parameters passed to it.

**TYPE: Listing 5.3. A function that uses the if and else statements.**

int GetMax( int nFirst, int nLast )

{

int nReturn;

if( nFirst > nLast )

nReturn = nFirst;

else

nReturn = nLast;

return nReturn;

}

70

**Using the switch Statement**

Sometimes you must choose between more than just one or two alternatives. Suppose you are implementing a simple menu function with three choices. If you use the if statement, you might wind up with a function like the one shown in Listing 5.4.

**TYPE: Listing 5.4. A menu-selection function.**

//

// Processes a selection from a character-based menu. If // a valid selection is made, the proper functions are // called, and true is returned. If an invalid selection // is made, false is returned.

bool HandleMenuSelection( char chSelection )

{

bool bValidSelection = true;

if( chSelection == `F' )

OpenNewFile();

else if( chSelection == `P' )

PrintDocument();

else if( chSelection == `S' )

SaveFile();

else

bValidSelection = false;

return bValidSelection;

}

This is already starting to look a little cluttered, but how bad would it look if you had a few more selections? What if you had 20 or 30? The solution is to use the switch statement. A switch statement evaluates an expression and then chooses from a list of choices, as shown in Listing 5.5.

**TYPE: Listing 5.5. Using the switch statement.**

bool HandleMenuSelection( char chSelection )

{

bool bValidSelection = true;

switch( chSelection )

{

71

case `F':

OpenNewFile();

break;

case `P':

PrintDocument();

break;

case `S':

SaveFile();

break;

default:

bValidSelection = false;

}

return bValidSelection;

}

As Listing 5.5 shows, the switch statement has several different parts. Here are the major features of a switch statement:

• The switch() expression. The expression contained inside the switch parentheses is evaluated, and its value is used as the basis for making the selection.

• One or more case labels. Each case label includes a value. Every case label must be unique. If a case label's value matches the switch expression, the statements after the case label are executed. • One or more break statements. The break statement is used to stop execution inside a switch statement. A break statement is normally placed between every case. If a break statement is removed, statements in the next case are executed until a break is reached, or until no more statements remain inside the switch.

• A default label. The default label is selected when no case labels match the switch expression.

**What Is a Button?**

**New Term:** A *button* is a special type of window that contains a text or bitmap label, usually found in a dialog box, toolbar, or other window containing controls.

Five different types of buttons are provided by Windows:

• *Pushbuttons* have a raised, three-dimensional appearance and seem to be depressed as they are clicked with the mouse. Pushbuttons normally have a text label on the face of the control.

• *Radio buttons* consist of a round button with a label adjacent to it.

72

• *Check boxes* are made up of a square box that contains a check mark when selected and a label next to the control.

• *Owner-drawn* buttons are painted by the button's owner instead of by Windows. • *Group boxes* are simply rectangles that are used to surround other controls that have a common purpose.

In general, buttons are used to indicate a user selection. Buttons are used in Windows programs because they are convenient and easy for users to operate. Users have come to expect buttons to be presented in a large number of cases, especially when dialog boxes are present in a program.

**What Are Pushbuttons?**

Almost every dialog box has at least one pushbutton control to indicate actions that a user can invoke. Some common uses for pushbuttons include closing a dialog box, beginning a search, or asking for help.

**What Are Radio Buttons?**

Radio buttons are used when a selection must be made from several mutually exclusive options, such as a user's gender. Only one of the radio buttons, which usually are grouped together, is checked at any particular time.

**What Are Check Boxes?**

Check boxes are used as Boolean flags that indicate whether a particular condition is true or false. Unlike radio buttons, several check boxes in a group can be checked. Optionally, a check box can support a third state--disabled--meaning that the control is neither true nor false.

**What Are Group Boxes?**

A group box logically groups controls that are used for similar purposes. This helps the user understand the relationships between controls and makes a dialog box easier to use. Radio buttons are almost always enclosed in a group box so that it's obvious which controls are associated with each other.

**MFC Support for Buttons**

Button controls normally are created as part of a dialog box. After you add a button to a dialog box, you can use ClassWizard to add functions that can be used to handle events created when the button is pressed,

73

checked, or selected. You also use ClassWizard to create CButton objects that are associated with individual button controls.

You can use the MFC class CButton to interact with button controls--both buttons that have been added to a dialog box resource and buttons that have been created dynamically. Use ClassWizard to associate a button control with a specific CButton object.

**A Sample Project Using Buttons**

In order to see how button controls can be used with dialog boxes, create a dialog box-based project named Button using AppWizard, following the steps provided in Hour 4, "Dialog Boxes and C++ Classes." You will use this project for the rest of this hour as an example of how to use buttons in a dialog box.

Click the ResourceView tab in the project workspace. Open the dialog box editor by double-clicking the IDD\_BUTTON\_DIALOG icon in the Dialog resource folder.

The IDD\_BUTTON\_DIALOG dialog box is displayed in the dialog box editor, along with a dockable control toolbar or palette. The floating control palette contains all the controls available for a dialog box, as shown in Figure 5.1.



**Figure 5.1.** The floating control palette, showing the buttons and boxes needed to create basic dialog boxes.

There are four different icons on the control palette for buttons, each used for a particular button type. Use one of the following steps to add a button control to a dialog box:

• Drag a button control from the palette to the dialog box by pressing the left mouse button while over the control button, then dragging the mouse cursor to the dialog box with the left mouse button still pressed. Release the mouse button when the cursor is over the desired spot in the dialog box.

• Select a button control by clicking a control in the control palette. Click the desired location for the control in the dialog box, and the dialog box editor creates a control for you in that location.

74

These steps apply for all controls in the control palette. After you've added a control to the dialog box, you can use the mouse to reposition and resize it.

As a demonstration, add several buttons to the main dialog box used in the Button project. You will use these controls later this hour to demonstrate button events. Refer to Figure 5.2 for the location of the added buttons.



**Figure 5.2.** The main dialog box used by the Button project.

A total of five buttons are added to IDD\_BUTTON\_DIALOG. Use the values from Table 5.1 to set the properties for each control. Except for the ID and caption, all controls use the default set of properties.

**Table 5.1. Values used for controls in IDD\_BUTTON\_DIALOG.**

| **Control ID**  IDC\_BTN\_TEST  IDC\_RADIO\_HIGH IDC\_RADIO\_LOW  IDC\_GROUP\_VOLUME IDC\_CHECK\_AMP | **Button Type** | **Caption** |
| --- | --- | --- |
| Pushbutton | &Test |
| Radio button | &High |
| Radio button | &Low |
| Group control &Volume |  |
| Check box | &Amplified |

**Button Control Properties**

Like all controls, buttons have a set of properties that define the behavior of each control. Although there are four different types of button controls, they share a common set of properties. You can display the properties for a particular control by selecting Properties from the menu displayed when you right-click the control. These properties are shared by all button controls:

75

• ID: Used for the button's resource ID. A default resource ID, such as IDC\_BUTTON1, is supplied by Developer Studio. Using IDC\_ as a prefix for control resource IDs is a Microsoft naming convention.

• Caption: Indicates the text that appears as the button's label. Developer Studio supplies a default name, such as Button. To make one of the letters in the caption of a control the mnemonic key, precede it with an ampersand (&).

• Visible: Indicates that the button is initially visible. This check box is normally checked. • Disabled: Indicates that the button should be initially disabled. This check box is normally cleared. • Group: Marks the first control in a group. All controls following a control with this attribute are

considered part of the same group if this check box is cleared. A user can move between controls in the same group using the arrow keys.

• Tab Stop: Indicates that this control can be reached by pressing Tab on the keyboard. This check box is normally checked.

• Default Button: Marks this control as the dialog box's default button. There can be only one default button in a dialog box, and it is executed if the user presses Enter without using any other controls in the dialog box. This check box is normally cleared.

• Owner Draw: Indicates that the button will be drawn by the button's owner; in this case, the dialog box. In most cases, this check box is cleared.

Group boxes support the fewest properties of any button control. All button properties are supported except Default Button and Owner Draw.

Radio buttons do not use the default button property because they aren't used as default buttons. However, radio buttons do support two properties not used by pushbutton controls:

• Auto: Automatically changes the state of the control when it is selected. This check box is normally selected.

• Left Text: Places the control's label on the left side of the check box instead of the right. This check box is normally cleared.

Check boxes support the same properties as radio controls, except that they are used with one additional attribute:

• Tri-state: The check box can have three states instead of two. In addition to true and false, the control can be disabled, meaning that the value is neither true nor false.

In addition, all controls have a property page that is labeled Extended Styles. These styles are rarely used, and aren't discussed in this book.

76

**Using Standard Pushbutton Layouts in Your Dialog Boxes**

Several pushbuttons are commonly used in dialog boxes that contain controls. Because each of these pushbuttons carries a specific meaning, you should try to use the standard terminology whenever possible because it minimizes the amount of work required for users of your programs. Here are the standard meanings for these buttons:

• OK: Used to close and accept any information that is present in the dialog box. Any user-supplied information in the dialog box is used by the program. Note that the OK pushbutton is the only button spelled with all capital letters.

• Cancel: Used to close the dialog box and remove any changes that might have been performed while the dialog box was open. If there are changes that cannot be reversed, the label for this button should be changed to read Close. Changing the label for a button is discussed later in this hour.

• Close: Used to close the dialog box. It does not necessarily imply that any action is taken by the program. Close is most often used when a Cancel button cannot be used to remove changes made while the dialog box is open. Many programs change a Cancel button into a Close button. • Help: Used to request context-sensitive help for the open dialog box.

• Apply: Used to perform changes based on data that has been entered in the dialog box. Unlike the OK button, the dialog box should remain open after the Apply button is pressed.

**Binding a Button Control to a CButton Object**

The easiest way to set or retrieve the value of a control is to associate it with a class-member variable using ClassWizard. When associating a member variable with a control, you can associate the member variable either with the control or with the control's value. Member variables representing buttons are rarely associated by value; instead, the CButton class is used to represent most button controls. You will learn about associating member variables by value with dialog box controls in Hour 6, "Using Edit Controls."

To add a member variable to a CDialog-derived class, follow these steps:

1. Open ClassWizard.

2. Select the tab labeled Member Variables.

3. Select the CDialog-derived class that manages the dialog box; in this case, CButtonDlg. 4. Select the control ID representing the control associated with the new member variable.

5. Press the button labeled Add Variable. An Add Member Variable dialog box appears. Enter the control's name, category, and variable type, then press OK.

6. Close ClassWizard.

77

Follow these steps for all controls added to the IDD\_BUTTON\_DIALOG earlier. Use the values from Table 5.2 for each new member variable added to CButtonDlg.

**Table 5.2. Values used to add member variables for CButtonDlg.**

| **Control ID**  IDC\_BTN\_TEST  IDC\_GROUP\_VOLUME m\_btnVolume IDC\_CHECK\_AMP | **Variable Name** | **Category Type** |  |
| --- | --- | --- | --- |
| m\_btnTest | Control | CButton |
|  | Control | CButton |
| m\_btnAmp | Control | CButton |

ClassWizard automatically adds the member variables to the CButtonDlg class declaration for you.

**Adding Button Events to a Dialog Box Class**

Although the buttons are part of the dialog box resource and appear whenever the dialog box is displayed, nothing happens when the buttons are used because no button events are handled by the dialog box class.

Pushbuttons are normally associated with button events in a dialog box class. To add a button event for IDC\_BTN\_TEST, follow these steps:

1. Open ClassWizard.

2. Select the tab labeled Message Maps.

3. Select CButtonDlg as the class name.

4. Select IDC\_BTN\_TEST as the object ID.

5. Select BN\_CLICKED from the Messages list box.

6. Press the button labeled Add Function and accept the default name for the member function. 7. Close ClassWizard.

Check boxes and radio buttons sometimes use BN\_CLICKED messages, but not as often as pushbuttons. Add the source code from Listing 5.6 to the CButtonDlg::OnBtnTest function, then compile and run the project.

**TYPE: Listing 5.6. The CButtonDlg::OnBtnTest member function.**

void CButtonDlg::OnBtnTest()

{

AfxMessageBox( "Test button pressed" );

78

}

**Changing a Button's Label**

Like all controls, a button is a just a special type of window. For that reason, the MFC class library uses the CWnd class as a base class for all control classes. To change the label for a button, you can use the SetWindowText function.

This function commonly is used to change the label for buttons after the dialog box has been created. You can use the SetWindowText function to change the Amplify button from the earlier example into a Record button. To do so, replace the CButtonDlg::OnBtnTest function with the function provided in Listing 5.7.

**TYPE: Listing 5.7. Changing the label for several buttons.**

void CButtonDlg::OnBtnTest()

{

static BOOL bSetWaterLevel = TRUE;

if( bSetWaterLevel == TRUE )

{

m\_btnVolume.SetWindowText( "&Water Level" );

m\_btnAmp.SetWindowText( "&Record" );

bSetWaterLevel = FALSE;

}

else

{

m\_btnVolume.SetWindowText( "&Volume" );

m\_btnAmp.SetWindowText( "&Amplify" );

bSetWaterLevel = TRUE;

}

}

After you build the Button example using the code from Listing 5.7, the radio button group will alternate between Volume and Water Level.

**Enabling and Disabling Buttons**

Most controls are enabled by default, although a control can be initially disabled by setting that attribute in its property list. A control can be selected only if it is enabled. The CWnd class includes the EnableWindow member function that allows a CWnd object to be enabled or disabled. Because CButton and all other

79

control classes are derived from CWnd, they include all the member data and member functions from the CWnd class, and you can disable a button like this:

pButton->EnableWindow( FALSE ); // Disables control

The parameter for EnableWindow is TRUE if the window or control should be enabled, and FALSE if it should be disabled. The default parameter for EnableWindow sets the parameter to TRUE because no parameter is needed to enable the control:

pButton->EnableWindow(); // Enables control

It is common practice for buttons and other controls to be enabled or disabled based on events that are received by the dialog box. As an example, pressing one button can cause another button to be disabled or enabled. To disable a dialog box control, replace the CButtonDlg::OnBtnTest function with the source code provided in Listing 5.8.

**TYPE: Listing 5.8. Using CWnd::EnableWindow to disable a dialog box control.**

void CButtonDlg::OnBtnTest()

{

static BOOL bEnableControl = FALSE;

m\_btnAmp.EnableWindow( bEnableControl );

if( bEnableControl == TRUE )

bEnableControl = FALSE;

else

bEnableControl = TRUE;

}

Now when you click the Test button, the Amplify check box is disabled. When you click the Test button again, the check box is enabled.

**Hiding a Button**

It's not unusual to need to hide a button that is located in a dialog box. Often, a button has its properties set to be hidden by default. Once again, the CWnd class has a member function that can be used to hide or display a window as needed. Use the CWnd::ShowWindow member function like this:

pButton->ShowWindow( SW\_HIDE ); // Hide control

80